

SOFTWARE

KR C...

Expert Programming

KUKA System Software (KSS)

Release 4.1

© Copyright **KUKA Roboter GmbH**

This documentation or excerpts thereof may not be reproduced or disclosed to third parties without the express permission of the publishers. Other functions not described in this documentation may be operable in the controller. The user has no claim to these functions, however, in the case of a replacement or service work.

We have checked the content of this documentation for conformity with the hardware and software described. Nevertheless, discrepancies cannot be precluded, for which reason we are not able to guarantee total conformity. The information in this documentation is checked on a regular basis, however, and necessary corrections will be incorporated in subsequent editions.

Subject to technical alterations without an effect on the function.

PD Interleaf

Contents

1	General information on KRL programs	7
1.1	Structure and creation of programs	7
1.1.1	Program interface	7
1.1.2	File concept	9
1.1.3	File structure	9
1.2	Creating and editing programs	11
1.2.1	Creating a new program	11
1.2.2	Editing, compiling and linking a program	12
1.3	Altering programs	14
1.3.1	Program correction	14
1.3.2	Editor	14
1.3.2.1	Block functions	14
1.3.2.2	Copy (CTRL-C)	14
1.3.2.3	Paste (CTRL-V)	15
1.3.2.4	Cut (CTRL-X)	15
1.3.2.5	Delete	15
1.3.2.6	Find	16
1.3.2.7	Replace	16
1.4	Hiding program sections	19
1.4.1	FOLD	19
1.4.1.1	Example program	20
1.5	Program run modes	22
1.6	Error treatment	24
1.7	Comments	27
2	Variables and declarations	29
2.1	Variables and names	29
2.2	Data objects	31
2.2.1	Declaration and initialization of data objects	31
2.2.2	Simple data types	33
2.2.3	Arrays	35
2.2.4	Character strings	38
2.2.5	Structures	38
2.2.6	Enumeration types	40
2.3	Data manipulation	42
2.3.1	Operators	42
2.3.1.1	Arithmetic operators	42
2.3.1.2	Geometric operator	43
2.3.1.3	Relational operators	47
2.3.1.4	Logical operators	48
2.3.1.5	Bit operators	49
2.3.1.6	Priority of operators	51
2.3.2	Standard functions	52
2.4	System variables and system files	54
3	Motion programming	59
3.1	Application of the various coordinate systems	59

3.2	Point-to-point motions (PTP)	66
3.2.1	General (Synchronous PTP)	66
3.2.2	Higher motion profile	67
3.2.3	Motion commands	68
3.3	Continuous-path motions (CP motions = Continuous Path)	77
3.3.1	Velocity and acceleration	77
3.3.2	Orientation control	78
3.3.3	Linear motions	83
3.3.4	Circular motions	84
3.4	Computer advance run	86
3.5	Motions with approximate positioning	89
3.5.1	PTP-PTP approximate positioning	90
3.5.2	LIN-LIN approximate positioning	93
3.5.3	CIRC-CIRC and CIRC-LIN approximate positioning	96
3.5.4	PTP-CP approximate positioning	99
3.5.5	Tool change during approximate positioning	102
3.6	Teaching points	103
3.7	Motion parameters	104
4	KRL assistant	105
4.1	Position specifications	106
4.2	[PTP] positioning	109
4.3	[LIN] linear motion	111
4.4	[CIRC] circular motion	113
5	Program execution control	115
5.1	Program branches	115
5.1.1	Jump instruction	115
5.1.2	Conditional branch	116
5.1.3	Switch	117
5.2	Loops	118
5.2.1	Counting loop	118
5.2.2	Rejecting loop	120
5.2.3	Non-rejecting loop	121
5.2.4	Endless loop	123
5.2.5	Premature termination of loop execution	123
5.3	Wait instructions	124
5.3.1	Waiting for an event	124
5.3.2	Wait times	124
5.4	Stopping the program	125
5.5	Confirming messages	126
6	Input/output instructions	127
6.1	General	127
6.2	Binary inputs/outputs	128
6.3	Digital inputs/outputs	131

6.3.1	Signal declaration	131
6.3.2	Setting outputs at the end point	133
6.4	Pulse outputs	136
6.5	Analog inputs/outputs	138
6.5.1	Analog outputs	138
6.5.2	Analog inputs	141
6.6	Predefined digital inputs	143
7	Subprograms and functions	145
7.1	Declaration	145
7.2	Subprogram and function call and parameter transfer	148
8	Interrupt handling	153
8.1	Declaration	154
8.2	Activating interrupts	156
8.3	Stopping active motions	160
8.4	Cancelling interrupt routines	161
8.5	Use of cyclical flags	164
9	Trigger – Path-related switching actions	165
9.1	Switching action at the start or end point of the path	165
9.2	Switching action at any point on the path	169
9.3	Tips and tricks	174
9.3.1	Overlapping Trigger statements	174
10	Data lists	175
10.1	Local data lists	175
10.2	Global data lists	176
11	External editor	179
11.1	Starting the external editor	180
11.2	Operator control	182
11.3	“File” menu	185
11.3.1	Open	185
11.3.2	Save	185
11.3.3	Print	186
11.3.4	Close file	186
11.3.5	Exit	187
11.4	“Edit” menu	188
11.4.1	Cut (“CTRL”+“X”)	188
11.4.2	Copy (“CTRL”+“C”)	188
11.4.3	Paste as	188
11.4.4	Delete	189
11.4.5	Select all	189
11.5	“Util” menu	190

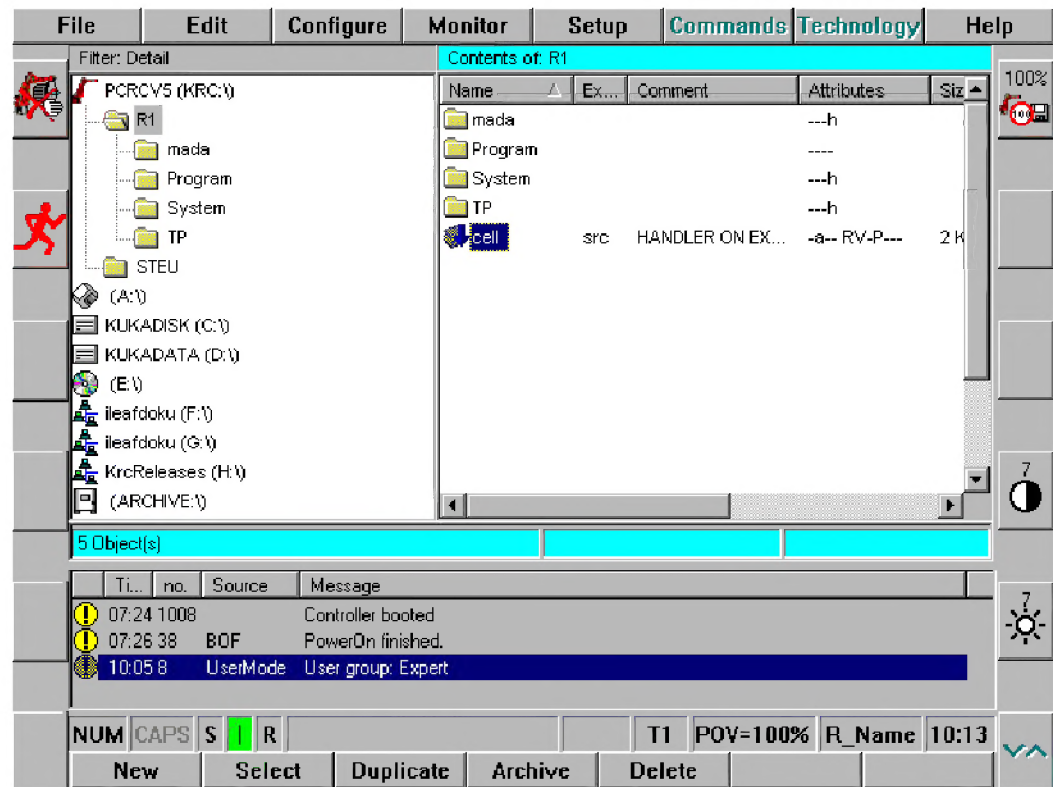
11.5.1	Mirror	190
11.5.2	Manual entry	191
11.5.3	Block change	195
11.5.4	Clean data list	195
11.5.5	TCP and Frame adjust	196
11.6	“HotEdit” menu	197
11.6.1	Base, TCP and World	197
11.6.2	Limits	199
11.6.3	TTS (correction coordinate system)	200
11.6.3.1	Position – TTS	200
11.6.3.2	Limits–TTS	204
11.7	“ExtUtil” menu	205
11.7.1	File – Mirror	205
11.7.2	File – Shift	208
11.7.2.1	Use existing reference file	209
11.7.2.2	Create new reference file	211
11.7.3	Setting the software limit switches	212
11.8	“Option” menu	213
11.8.1	Output setting	213
11.9	“Help” menu	215
11.9.1	Version	215
11.9.2	Stay on top	215

1 General information on KRL programs

1.1 Structure and creation of programs

1.1.1 Program interface

Switching to the expert level causes the user interface to change as illustrated below:



Whereas all the system files are invisible to the user, they can now be seen and also edited by the expert in the program window. Not only the file names and comments are displayed at expert level but also the file extensions, attributes and sizes.

The screenshot above shows the files and directories in the path "R1" displayed in the program window.

The files listed below are available as standard after the KR C1 software has been installed in the directory "KRC:\R1\MADA\":

File	Meaning
\$MACHINE.DAT	System data list with system variables for adapting the controller and the robot
\$ROBCOR.DAT	System data list with data for the dynamic model of the robot
MACHINE.UPG	System file for future upgrades
ROBCOR.UPG	System file for future upgrades

The following files can be found in the directory "KRC:\R1\SYSTEM\":

File	Meaning
\$CONFIG.DAT	System data list with general configuration data
BAS.SRC	Basic package for motion control
IR_STOPM.SRC	Program for fault service functions in response to malfunctions
SPS.SUB	Submit file for parallel monitoring

The following files can be found in the directory "KRC:\R1\TP\":

File	Meaning
A10.DAT A10.SRC	Application technology package for arc welding with analog reference voltages
A10_INI.DAT A10_INI.SRC	Application technology package for initializing arc welding with analog reference voltages
A20.DAT A20.SRC	Application technology package for arc welding with digital program numbers
A50.DAT A50.SRC	Application technology package for use of the LIBO (through-the-arc) sensor
ARC_MSG.SRC	Program for programming messages for arc welding
ARCSPS.SUB	Submit file for arc welding
BOSCH.SRC	Program for spot welding with serial interface to Bosch spot welding timer PSS5200.521C
COR_T1.SRC	Tool correction program (old version)
CORRTOOL.DAT CORRTOOL.SRC	Tool correction program
FLT_SERV.DAT FLT_SERV.SRC	Program for user-defined fault service functions in arc welding
H50.SRC	Gripper package
H70.SRC	Touch sensor package
MSG_DEMO.SRC	Program with examples of user messages
NEW_SERV.SRC	Program for altering error reactions for FLT_SERV
P00.DAT P00.SRC	Program package for coupling with a PLC

PERCEPT.SRC	Program for calling the PERCEPTRON protocol
USER_GRP.DAT USER_GRP.SRC	Program for user-defined gripper control
USERSPOT.DAT USERSPOT.SRC	Program package for user-defined spot welding
WEAV_DEF.SRC	Program for weave motions with arc welding

The following file can be found in the directory “KRC:\R1\”:

CELL.SRC	Program for controlling robots via a central PLC. Here, an application program is selected by means of a program number
----------	---

1.1.2 File concept

A KRL program can be made up of SRC and DAT files.



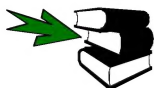
Further information on program creation can be found in this chapter in the section **[Creating and editing programs]**.

The “SRC” file contains the actual program code. There are two variants: DEF and DEFFCT (with return value). The “DAT” file, on the other hand, contains the specific program data. This division is based on the KRL file concept: apart from the processing sequence, the program contains various actions which the industrial robot is to perform. These can be special motion sequences, the opening or closing of a gripper, or complex sequences, such as the control of a welding gun taking the related constraints into consideration.

For the purpose of testing programs, it is helpful and/or necessary to be able to execute tasks of this nature individually. The KRL file concept is ideally suited to the special requirements of robot programming.

1.1.3 File structure

A file is the unit that is programmed by the programmer. It thus corresponds to a file on the hard disk or in the memory (RAM). Any program in KRL may consist of one or more files. Simple programs contain exactly one file. More complex tasks can be solved better using a program that consists of several files.



Detailed information on subprograms and functions can be found in the chapter **[Subprograms and functions]**.



The inner structure of a KRL file comprises the declaration section, the instruction section and up to 255 local subprograms and functions

DEF

The object name without an extension is also the name of the file and is therefore prefixed by “DEF” in the declaration. The name may consist of up to 24 characters and must not be a keyword (see Section **[Variables and declarations]**). Every file begins with the declaration “DEF” and ends with “END”.

```
DEF NAME (x1:IN)
Declarations
Instructions
END
```

Declaration Declarations are already evaluated before program execution, i.e. during compilation. No instructions may therefore be located in the declaration section. The first instruction is the beginning of the instruction section.

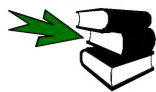
Instruction Unlike declarations, instructions are of a dynamic nature. They are executed when the program is processed.

Data list A robot program can consist of just a single program file or a program file with related data list. The data list and file are identified as belonging together by their common name. The names differ in their extension only, e.g.:

File: **PROG1.SRC**
Data list: **PROG1.DAT**



Only value assignments with “=” instructions are allowed in data lists. If the data list and the file have the same name, variables declared in the data list can be used in the same way as variables declared in the SRC file.



Detailed information can be found in the chapter **[Data lists]**.

1.2 Creating and editing programs

1.2.1 Creating a new program

New

As a robot program can also be written without a data list, the file and data list are not both automatically created at the same time at expert level. To create a new program, press the softkey "New". The following window will be opened:

Template selection	
Filename	Filter comment
Cell	Automatik extern dispatcher
Expert	Expert module
Expert Submit	Expert submit
Function	Function
Module	Module
Submit	User submit

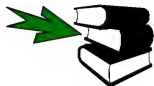
Please choose a template:

OK

You are prompted to select a template. Do this using the arrow keys and confirm it by pressing the softkey "OK" or the Enter key.



The available templates cannot be freely created in all directories.



Further information on templates can be found in the **Operating Handbook** in the documentation **Operator Control**, chapter **[Navigator]**, section **[Appendix]**.

The individual templates:

Module:

An SRC file and a DAT file are created containing a skeleton program.

Expert:

An SRC file and a DAT file are created containing merely the header DEF... and END.

Cell:

Here, only an SRC file containing a skeleton program is created. This program is used for controlling the robot via a central PLC.

Function:

Here, a function (SRC file) is created containing the header DEF... and END.

Submit:

A SUB file with a skeleton program is created. The Submit file contains instructions and can be used, for example, for cyclical monitoring (grippers, etc.). The Submit file works parallel to the robot and is processed by the controller interpreter.


Expert Submit:




As with the Submit template, a SUB file is created, this time containing merely the header DEF... and END.



The header DEF... and END and the skeleton programs of the individual templates are located, for the template Cell, for example, in "C:\KRC\ROBOTER\TEMPLATE\CellVorgabe.src".

Once you have selected the corresponding template, you are prompted to enter a name for the file created.

Name	Ext...	Comment	Attributes
 <input type="text"/>	<input type="text" value="---"/>	<input type="text"/>	<input type="text" value="---"/>

 File name
(max. 24 characters)
  File extension
(SRC, DAT or SUB)
  Comment

Only the file name is of vital importance and may be up to 24 characters long. The file extension is added automatically. If you wish to add a comment, move the cursor to the corresponding box using the right arrow key and enter the desired text.

OK

Press the softkey "OK" or the Enter key to acknowledge these entries.



The data list is mandatory if you also want to insert menu-driven commands in your SRC file.

1.2.2 Editing, compiling and linking a program

After you have created a file or data list by means of "New", you can edit them using the editor. The softkey "Edit" is used for this purpose. On closing the editor, the complete program code is compiled, i.e. the textual KRL code is translated into a machine language that can be understood by the controller.



In order to retain the clarity of the program, branches, for example, must be indented at several levels. In the editor, this can be done using the space-bar.

Compiler

In this process, the compiler checks that the code is syntactically and semantically correct. If errors are detected, a corresponding message is generated and an error file created with the file extension ".ERR".



Only programs that contain no errors can be selected and executed.



Further information on handling editing errors can be found in the section **[Error treatment]**.

Linkage editor

On loading a program via the softkey "Select", all the files and data lists required are linked to create a program. During linking, it is checked whether all the modules are present, compiled and free from errors. When transferring parameters, the linkage editor also checks the type compatibility of the transfer parameters. If errors occur during linking, an error file with the extension ".ERR" is created, as in compilation.



You can also write a KRL program using any normal text editor and then load it into the system memory by means of the softkey "Load". In this case, however, you must make sure yourself that all the necessary initializations (e.g. axis velocities) are carried out.

The following is an example of a simple program for defining axis velocities and accelerations:



DEF PROG1()

;----- Declaration section -----
INT J

;----- Instruction section -----
\$VEL_AXIS[1]=100 ;Specification of axis velocities
\$VEL_AXIS[2]=100
\$VEL_AXIS[3]=100
\$VEL_AXIS[4]=100
\$VEL_AXIS[5]=100
\$VEL_AXIS[6]=100

\$ACC_AXIS[1]=100 ;Specification of axis accelerations
\$ACC_AXIS[2]=100
\$ACC_AXIS[3]=100
\$ACC_AXIS[4]=100
\$ACC_AXIS[5]=100
\$ACC_AXIS[6]=100

PTP {A1 0,A2 -90,A3 90,A4 0,A5 0,A6 0}

FOR J=1 TO 5
PTP {A1 4}
PTP {A2 -7,A3 5}
PTP {A1 0,A2 -9,A3 9}
ENDFOR

PTP {A1 0,A2 -90,A3 90,A4 0,A5 0,A6 0}
END

NOTES:

1.3 Altering programs

There are basically two methods of altering a program at the Expert level of the graphical user interface:

- Program correction (PROKOR)
- Editor

1.3.1 Program correction

Program correction is the standard method. The PROCOR mode is automatically active when a program is selected or a running program is stopped.

Here, you can enter or edit commands that affect just **one** program line – i.e. no check structures (loops etc.) or variable declarations – using the inline form or ASCII code (at expert level).



If incorrect entries are selected, these are immediately deleted when the program line is left and an error message appears in the message window.

1.3.2 Editor

If you want to edit or insert certain KRL commands or program structures, the editor therefore has to be used. Since the complete code is compiled when the editor is closed, errors can also be detected which only occur in the interaction of several lines (e.g. incorrectly declared variables).

1.3.2.1 Block functions



These functions are only available in the editor at the “Expert” user level. You must open a program, whose contents you wish to change with the help of the block functions, using the softkey “Edit”. How you first switch to the “Expert” user level is described in the documentation **[Configuring the system]**, in the section “User levels”.

First position the blinking edit cursor at the start or end of the program section that is to be moved. Then hold down the “Shift” key on the keyboard while you move the cursor up or down. In this way you select a program section that can then be edited using the block functions in the next procedure. The selected section can be recognized by the color highlight.



Press the menu key “Program” and select the desired function from the menu that is opened.



If the keyboard and numeric keypad are used for the block functions, the NUM function must be deactivated. This is done by pressing the “Num” key on the keypad. The corresponding display in the status line is then switched off.

1.3.2.2 Copy (CTRL-C)

The selected program section is copied to the clipboard for further editing. It can subsequently be inserted elsewhere.

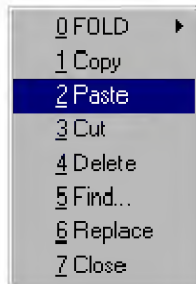




Alternatively, you can hold down the CTRL key in the numeric keypad and press the C key on the keyboard. Then release both keys.

1.3.2.3 Paste (CTRL-V)

Move the edit cursor to the position where the program section previously “cut” or “copied” is to be reinserted.



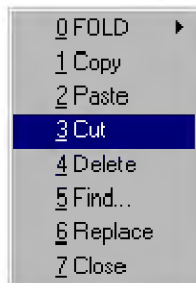
Now select the option “Block paste”. The previously selected program section is inserted below the edit cursor.



Alternatively, you can hold down the CTRL key in the numeric keypad and press the V key on the keyboard. Then release both keys.

1.3.2.4 Cut (CTRL-X)

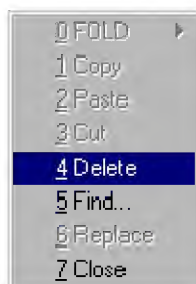
If you select the “Block cut” option from the menu, the selected program section is copied to the clipboard and deleted from the program listing.



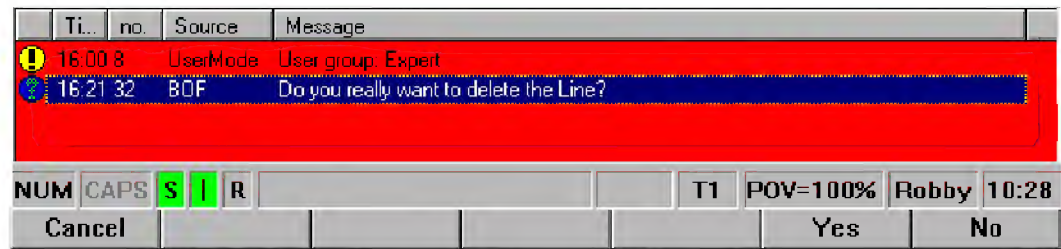
Alternatively, you can hold down the CTRL key in the numeric keypad and press the X key on the keyboard. Then release both keys.

1.3.2.5 Delete

The selected area can be removed from the program. It is not copied to the clipboard in this case. The deleted program section is thus lost irretrievably.



For this reason, a request for confirmation, which must be answered via the softkey bar, is generated in the message window.

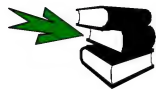


- **Cancel** The “Delete” action is cancelled;
- **Yes** The selected area is irrevocably deleted;
- **No** The “Delete” function is cancelled.



If you select the “Delete” option from the menu, the selected program section is deleted from the program listing without being copied to the clipboard.

1.3.2.6 Find

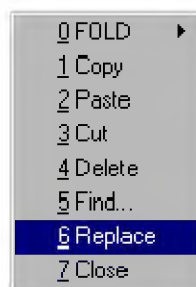


Further information can be found in the **Operating Handbook** in the documentation **User programming**, chapter **[Program editing]**, section **[Working with the program editor]**.

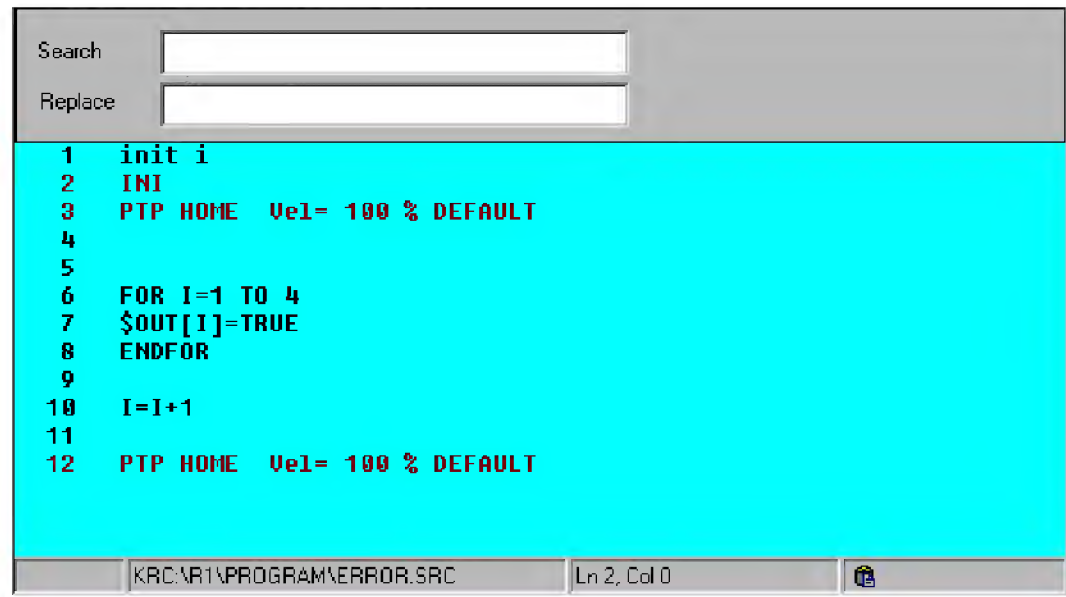
1.3.2.7 Replace

The “Find and replace” function is only available at the expert level, and there only in the editor. This function searches the visible sections of the program (not Fold lines or opened Folds) for one specified character string and enables it to be substituted by another defined character string.

This is done by selecting the option “Replace” from the “Program” menu.



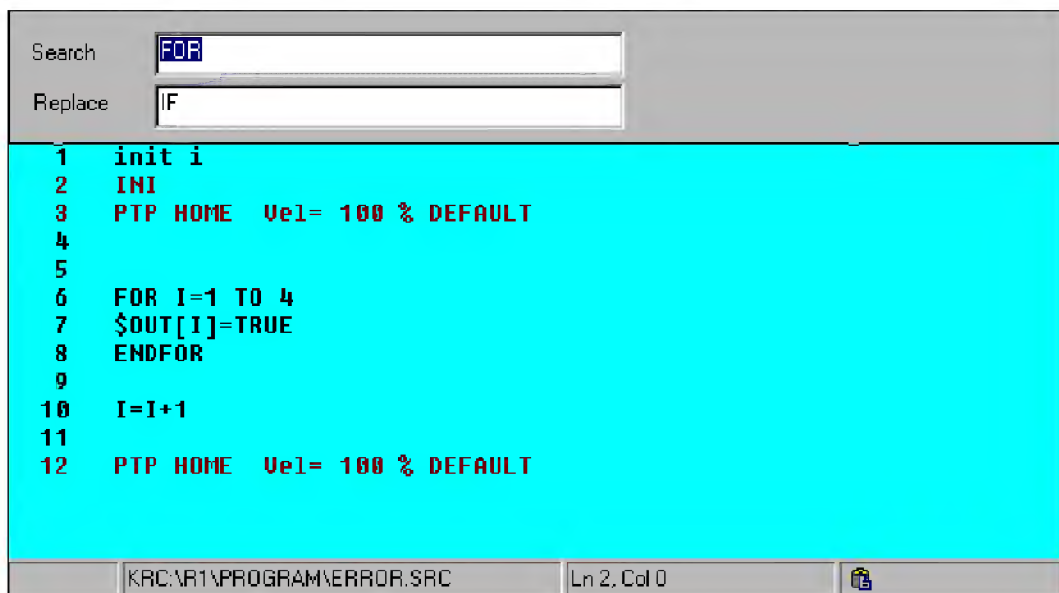
The following window is opened:



The softkey bar changes.



Enter a character string in the Search line and move down to the Replace line using the arrow key. Enter here the term that is to replace the search string.



Find

replace

If the term being searched for occurs more than once in the document and you wish to replace it in just one particular place, keep pressing the softkey “Find” until you have found the place in question. Then press “replace”. The search string is then replaced by the term specified.

If you wish to replace all occurrences of the search string in the program, or in a highlighted section thereof, enter the replacement term in the Search/Replace box as described above and press the softkey “repl. All”.

The following message appears in the message window “The specified or marked region has been searched.” (confirmation that the entire program or the marked section has been

Cancel

searched). When the softkey "Cancel" is pressed, the Replace function is terminated and the number of replacements made since activation of this function is displayed in the message window.

	Time	no.	Source	Message
!	16:38 40		BOF	The specified or marked region has been searched.
!	16:39 39		BOF	2 replacements made



NOTES:

1.4 Hiding program sections

Unlike normal editors, the KCP editor allows a requirement-specific display of the program contents. The user, for example, only sees the important contents of a program, while at expert level the whole program is visible.

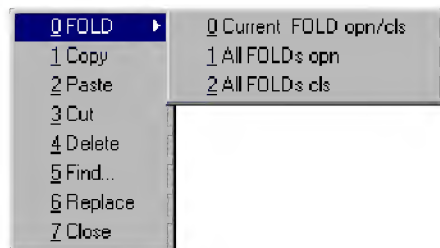
1.4.1 FOLD

The KUKA user interface uses a special technique to display a program clearly. Instructions in the form of KRL comments make it possible to suppress the display of subsequent parts of the program. In this way the program is subdivided into meaningful sections, called “FOLDS” due to their folder-like nature.

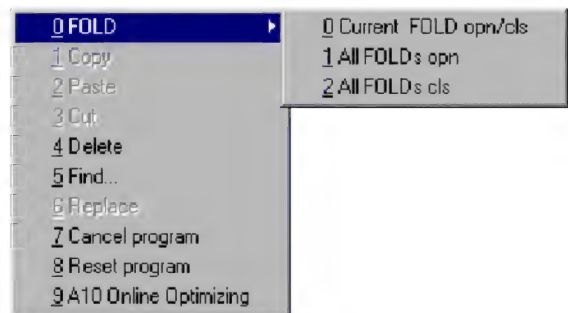
“FOLDS” are “closed” by default and can only be “opened” at expert level. You then obtain information which is invisible to the user on the KUKA graphical user interface (KUKA GUI). At expert level you have the possibility of making a KRL block invisible at user level. This is done by enclosing the relevant declarations or instructions within the designations “**; FOLD**” and “**; ENDFOLD**”.

Program

Folds in a program can be displayed or hidden by pressing the menu key “Program” and then selecting “FOLD” and the desired command.



Program in the editor



Program selected

The following options are available:

- **Current FOLD opn/cls** opens or closes the FOLD of the line in which the edit cursor is positioned
- **All FOLDs opn** opens all FOLDs of the program
- **All FOLDs cls** closes all FOLDs of the program



If a selected program with open Folds is reset, these Folds are automatically closed.

Of the sequence...

```
;FOLD RESET OUT

FOR I=1 TO 16
  $OUT[I]=FALSE
ENDFOR

;ENDFOLD
```

...only the words **"RESET OUT"** can be seen on the user interface with the Folds closed. With this command, for example, you can make the declaration and initialization sections invisible to the user.

1.4.1.1 Example program



```

DEF FOLDS()

;FOLD DECLARATION;% additional information
;----- Declaration section -----
EXT BAS (BAS_COMMAND :IN,REAL :IN )
DECL AXIS HOME
INT I
;ENDFOLD

;FOLD INITIALISATION
;----- Initialization -----
INTERRUPT DECL 3 WHEN $STOPMESS==TRUE DO IR_STOPM ( )
INTERRUPT ON 3
BAS (#INITMOV,0 ) ;Initialization of velocities,
                  ;Accelerations, $BASE, $TOOL, etc.
FOR I=1 TO 16
    $OUT[I]=FALSE
ENDFOR
HOME={AXIS: A1 0,A2 -90,A3 90,A4 0,A5 30,A6 0}
;ENDFOLD

;----- Main section -----
PTP HOME ;BCO run
LIN {X 540,Y 630,Z 1500,A 0,B 90,C 0}
PTP HOME

END

```

The example program has the following appearance on the screen:

```

1
2  DECLARATION
3
4  INITIALISATION
5
6  ;----- Hauptteil -----

```

The same program with the Folds open:


```

1
2  DECLARATION
3  ;----- Deklarationsteil -----
4  EXT BAS (BAS_COMMAND :IN,REAL :IN )
5  DECL AXIS HOME
6  INT I
7
8  INITIALISATION
9  ;----- Initialisierung -----
10 INTERRUPT DECL 3 WHEN $STOPMESS==TRUE DO IR_STOPM ( )
11 INTERRUPT ON 3
12 BAS (#INITMOV,0 ) ;Initialisierung von Geschwindigkeiten,
13 ;Beschleunigungen, $BASE, $TOOL, etc.
14 FOR I=1 TO 16
15 $OUT[I]=FALSE
16 ENDFOR
17 HOME={AXIS: A1 0,A2 -90,A3 90,A4 0,A5 0,A6 0}
18
19 ;----- Hauptteil -----

```

In the closed FOLD, only the expression after the keyword “FOLD” is visible. In the opened FOLD, on the other hand, all instructions and declarations can be seen.






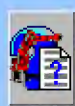
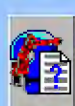
“FOLD” is merely an instruction for the editor. The compiler interprets the FOLD statements as normal comments because of the preceding semicolon.

1.5 Program run modes

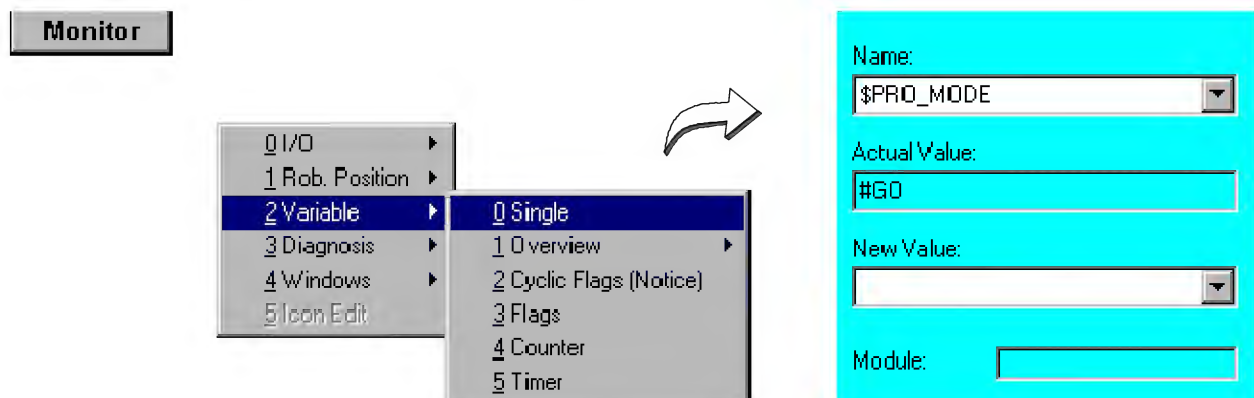
The program run modes define whether program execution is to take place

- without a program stop,
- motion instruction by motion instruction, or
- step by step.

All the program run modes are described in the following table.

Run mode	Description
GO 	All instructions in the program are executed up to the end of the program without a STOP.
MSTEP 	Motion Step (motion instruction) The program is executed one motion instruction at a time, i.e. with a STOP after each motion instruction. The program is executed without advance processing.
ISTEP 	Incremental Step (single step) The program is executed step by step, i.e. with a STOP after each instruction (including blank lines). The program is executed without advance processing.
PSTEP 	Program Step (program step) Subprograms are executed completely. The program is executed without advance processing.
CSTEP 	Continuous Step (motion instruction) The program is executed one motion instruction at a time, i.e. with a STOP after each motion instruction with exact positioning. The program is executed with advance processing, i.e. the points are approximated.

The program run modes GO, MSTEP and ISTEP can be selected on the KCP using a status key or via the variable "\$PRO_MODE". PSTEP and CSTEP, on the other hand, can only be set via the variable "\$PRO_MODE". In order to modify the state, activate the menu function "Monitor" -> "Variable" -> "Single". Then enter the variable "\$PRO_MODE" in the input box "Name" and the desired value in the box "New Value".





The program run modes “#PSTEP” and “#CSTEP” can only be selected via the variable modification function and not using the status keys.



More detailed information can be found in the chapter **[Variables and declarations]**, section **[Data objects]** under **(Enumeration types)**.



NOTES:

1.6 Error treatment



If an error occurs during compilation or linking, an error message is displayed and the file containing errors is indicated in the Navigator.



The file "ERROR.SRC", which was (incorrectly) created, serves as an example:

```

1  DEF  ERROR ( )
2  INI
3  PTP HOME  Vel= 100 % DEFAULT
4
5  FOR I=1 TO 4
6  $OUT[I]=TRUE
7  ENDFOR
8
9  I == I + 1
10
11 PTP HOME  Vel= 100 % DEFAULT
12 END

```

When the editor is closed, a notification message with the number of errors appears in the message window.

	Time	no.	Source	Message
!	14:49 8		UserMode	User group: Expert
!	14:51 1421	HPU	/R1/ERROR	3 Compilation error

At the same time, the affected files are marked with a red cross.

Name	Ext...	Comment	Attribu
 error	src		-a-- R
 error	dat		-a-- R

The following softkey bar is available:

New	View ERR	Open	Edit DAT	Delete		
-----	----------	------	----------	--------	--	--

The softkey "Open" loads the file into the editor, while the softkey "Edit DAT" opens the Dat file in the editor. If you wish to delete the files containing errors, press "Delete"; you can then create a new file by pressing "New".

View ERR

This is opened by pressing the softkey "View ERR".

ErrorView(error.SRC)			
Line	Col.	Error ...	Description
52	5	2263	Type of loop va...
53	6	2249	Expression not ...
56	6	2309	(' expected
*1			
FOR I=1 TO 4			
Type of loop variable not equal to INT			

Title bar with the name of the file

Brief description

Error number

Line and column numbers of the lines with errors

Source text line containing errors

Error description = brief description

The softkey bar changes:

				jump to	Refresh	Close
--	--	--	--	---------	---------	-------



NOTE *1

The line numbers displayed correspond to the absolute line numbers in the program as a normal ASCII editor would display them. In order for the line numbers in the error display to agree with those in the KCP, all Folds must be open and Detail view and the DEF lines must be active. This display, however, is somewhat lacking in clarity, as all information is available even though it is not required. Further information on Detail view and DEF lines can be found in the section **[Hiding program sections]**.

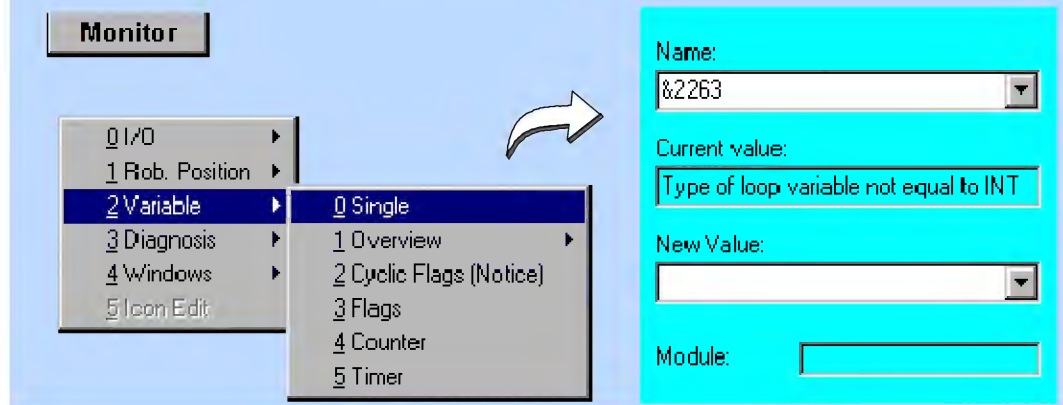
It is evident from the error display that the following errors have occurred:

- 3 lines in the SRC file contain errors;
- the line numbers of the lines with errors are 52, 53 and 56;
- in line 52 the error number
 - 2263: type of loop variable not equal to INT;
- in line 53 the error number
 - 2249: expression not equal to INT;
- in line 56 the error message
 - 2309: "(" character expected;

From error message "2263" it is readily evident that variable `I` has not been declared as an integer. Error message 2249 is also a result of the missing declaration, as the counter for a counting loop must always be of type INT. Message "2309" means: the compiler interprets the line as a subprogram call, in which, however, the brackets are missing.



You can display the meaning of the error numbers online using the menu function “Monitor” –> “Variable” –> “Modify”. To do this, enter the character “&” followed by the error number in the input box “Name” in the status window. In this case, for example, enter “&2263” and press the Enter key.



jump to

If you now load the SRC file (in this case “ERROR.SRC”) into the editor, you can make the appropriate corrections. This is made easier by the blinking cursor which positions itself in the first line containing errors. Make sure that the limited visibility is switched off and that the DEF-line is visible. Details can be found in the section **[Hiding program sections]**.

In the present example, the Folds do not have to be open. If you wish to open them, use the menu command “Program” –> “FOLD” –> “All FOLDS open”.



The line “INT I”, missing in the program initially created, must be inserted **before** the line “INI”. This is only possible if the line “DEF ERROR ()” is visible.

Correct the errors by inserting the line

INT I

before the INI line and deleting one of the equal signs in line 10.

I = I + 1

<pre> 1 DEF ERROR() 2 INT I 3 INI 4 PTP HOME Ve1= 100 % DEFAULT 5 6 FOR I=1 TO 4 7 \$OUT[I]=TRUE 8 ENDFOR 9 10 I=I+1 11 12 PTP HOME Ve1= 100 % DEFAULT 13 END </pre>	<p>Insert this line here</p> <p>Delete an equals sign</p>
--	---

Refresh

After closing the editor and saving the corrected file, you can press the softkey “Refresh” in the error list; if all errors have been eliminated, the error list disappears.

1.7 Comments

Comments are an important part of any computer program. They enables you to make your program transparent and also understandable for others. The execution speed of the program is not affected by comments.

Comments can be inserted at any point in a program. They are always preceded by a semi-colon “;”, e.g.:

```
...
PTP P1                ;Motion to start point
...
;--- Reset outputs ---
FOR I = 1 TO 16
    $OUT[I] = FALSE
ENDFOR
...
```


2 Variables and declarations

2.1 Variables and names

Beside the use of constants, in other words the direct specification of values in the form of numbers, symbols, etc., it is also possible to use variables and other forms of data in a KRL program.

In the programming of industrial robots, variables are required for the purpose of sensor processing, for example. They enable the value supplied by the sensor to be saved and evaluated at various points in the program. Arithmetic operations can also be performed in order to calculate a new position.

A variable is represented by a name in the program, this designation being freely selectable subject to certain restrictions.

Names

Names in KRL

- can have a maximum length of 24 characters,
- can consist of letters (A–Z), numbers (0–9) and the signs ‘_’ and ‘\$’,
- must not begin with a number,
- must not be a keyword.



As all system variables (see Section 2.4) begin with the ‘\$’ sign, this sign should not be used as the first character in self-defined names.

Examples of valid KRL names are

SENSOR_1

NOZZLE13

P1_TO_P12

A variable is to be regarded as a fixed memory area, whose contents can be addressed via the variable name. When the program is executed, the variable is therefore represented by a memory location (place) and a memory content (value).

Value
assignment

Values are then assigned to the variables using the equal sign (=). The statement

QUANTITY = 5

thus means that the value 5 is entered in the memory area with the address of QUANTITY. The exact address is of no interest to the programmer and is therefore assigned automatically by the compiler. It is only important that the memory content can be addressed in the program at all times with the aid of its name.

As different data objects (see Section 2.2) also have different memory requirements, the data type of a variable must be declared (see Section 2.2.1) before it is used.

Variable life The lifetime of a variable is the time during which the variable is allocated memory. It depends on whether the variable is declared in an SRC file or a data list:

- **Variable declared in an SRC file**

The lifetime is limited to the run time of the program. The memory area is deallocated again on completion of execution. The value of the variable is thus lost.

- **Variable declared in a data list (see chapter Data lists)**

The lifetime is independent of the run time of the program. The variable exists only as long as the data list exists. Such variables are therefore permanent (until the system is next switched off).



NOTES:

2.2 Data objects

Data objects are namable memory units of a particular data type. The memory units may consist of a different number of memory units (bytes, words, etc.). If such a data object is declared under a name by the programmer, a variable is created. The variable now occupies one or more memory locations, in which data can be written and read by the program. The symbolic naming of the memory locations with a freely selectable designation makes programming easier and more transparent and enhances the readability of the program.

The following example is intended to illustrate the term “data type”: A memory location with 8 bits contains the bit combination

```
00110101
```

How is this bit combination to be interpreted? Is it the binary notation of the number 53 or the ASCII character “5”, which is represented by the same bit pattern?

Data type

An important item of information is required in order to answer this question unambiguously, namely the specification of the data type of a data object. In the above case, this could be the type `INTEGER` or `CHARACTER`, for example.

Besides this computer-related reason for introducing data types, the programmer also benefits from the use of data types since it is possible to work with exactly the types that are particularly well suited to the specific application.

2.2.1 Declaration and initialization of data objects

DECL

Assignment of a variable name to a data type and reservation of the memory space are accomplished in KRL with the aid of the `DECL` declaration. By means of

```
DECL INT QUANTITY,NUMBER
```

you can declare, for example, two variables `QUANTITY` and `NUMBER` of the data type `INTEGER`.

The compiler thus knows these two variables and the associated data type and, when the variables are used, can check whether this data type permits the intended operation.

The declaration begins, as shown in the example, with the keyword `DECL`, followed by the data type and the list of variables that are to be assigned this data type.



When declaring variables and arrays of a predefined data type, the keyword `DECL` can be omitted. Besides the simple data types `INT`, `REAL`, `CHAR` and `BOOL` (see Section 2.2.2), the structure data types `POS`, `E6POS`, `FRAME`, `AXIS` and `E6AXIS` (see Section 2.2.5) are predefined, among others.

The declaration can be entirely omitted for variables (not arrays!) of the data type `POS`. The data type `POS` is the standard data type for variables.

The keyword `DECL` is indispensable in the declaration of freely definable structure or enumeration types (see Section 2.2.5 and 2.2.6).

Initialization

After a variable has been declared, its value is first set to invalid since it would otherwise depend on the random memory allocation. To make it possible to work with the variable, it must therefore be preallocated a specific value. This first value assignment to a variable is called initialization.



When creating new files by means of the softkey “New” on the KUKA user interface, an `INI` sequence is also automatically generated. The declaration of variables must always take place before this sequence.

A value assignment to a variable is an instruction and must therefore never be located in the declaration section. Initialization, however, can take place in the instruction section at any time. All declared variables should nevertheless ideally be initialized in an initialization section directly after the declaration section (see Fig. 1).

Only in data lists is it permissible to initialize variables directly in the declaration line.

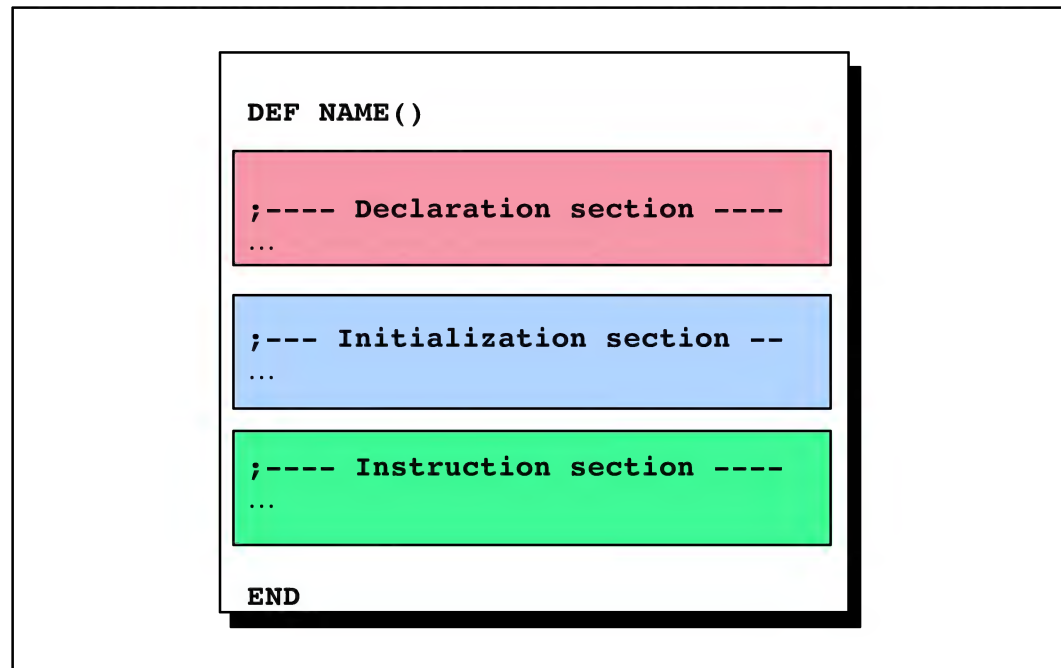


Fig. 1 Basic structure of a robot program



Further information can be found in the chapter **[Data lists]**.

2.2.2 Simple data types

By simple data types, we mean a number of basic data types that are available in most programming languages. In contrast to the structured data types (see sections 2.2.3–2.2.6), simple data types contain just one single value. The data types known in KRL are listed in Table 1 together with their respective ranges of values.

Data type	Integer	Real	Boolean	Character
Keyword	INT	REAL	BOOL	CHAR
Meaning	Integer	Floating-point number	Logic state	1 character
Range of values	$-2^{31} \dots 2^{31}-1$	$+1.1\text{E}-38 \dots \pm 3.4\text{E}+38$	TRUE, FALSE	ASCII character

Table 1 Simple data types

INT

The data type Integer is a subset of the set of integers. It can only be a subset because no computer can render the theoretically infinite set of integers. The 32 bits provided in the KR C... for integer types therefore result in 2^{31} integers plus signs. The number 0 counts as a positive number.

By means of

```
NUMBER = -23456
```

the variable NUMBER is assigned the value -23456.

If you assign an INTEGER variable a REAL value, the value will be rounded according to general rules (x.0 to x.49 rounded down, x.5 to x.99 rounded up). By means of the statement

```
NUMBER = 45.78
```

the INTEGER variable NUMBER is assigned the value 46.



Exception: The result of integer division is cut off at the decimal point, e.g.:
 $7/4 = 1$

Binary system
Hexadecimal
system

Whereas people calculate and think in the decimal system, a computer only knows zeros and ones, which are represented by the two states off and on. A state (off or on) is thus represented by a bit. For reasons of speed, the computer generally accesses a whole bundle of such zeros and ones. Typical bundle sizes are 8 bits (= 1 byte), 16 bits or 32 bits. For computer-oriented operations, representation in the binary system (number system to the base two using the digits 0 and 1) or in the hexadecimal system (number system to the base 16 using the characters 0–9 and A–F) is useful. Binary or hexadecimal integers can be specified in KRL with the aid of inverted commas (') and the prefix B for binary notation or H for hexadecimal notation.

D	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
H	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10

Table 2 The first 17 numbers in the decimal and hexadecimal systems

In KRL, you can therefore assign the number 90 to an integer variable in three different ways:

```

INTEG = 90                ;Decimal system
INTEG = 'B1011010'       ;Binary system
INTEG = 'H5A'            ;Hexadecimal system

```

Bin → Dec

Binary numbers are converted to the decimal system as follows:

1	0	1	1	0	1	0	$= 1 \cdot 2^6 + 0 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 90$
2^6	2^5	2^4	2^3	2^2	2^1	2^0	

Hex → Dec

To transfer numbers from the hexadecimal system to the decimal system, proceed as follows:

5	A	$= 5 \cdot 16^1 + 10 \cdot 16^0 = 90$
16^1	16^0	

REAL

In floating-point representation, a number is divided into a fixed-point part and an exponent and represented in standardized form. This results in the following representations, for example:

```

    5.3      as    0.53000000 E+01
   -100      as    -0.10000000 E+03
    0.0513   as    0.51300000 E-01

```

When calculating with real values, it must be borne in mind that because of the limited number of places after the floating point and the inherent inaccuracy, the usual algebraic laws are no longer applicable in all cases. By the laws of algebra, for example:

$$\frac{1}{3} \times 3 = 1$$

If a computer performs this calculation, it could produce a result of just 0.99999999 E+00. A logic comparison of this number with the number 1 would result in the value FALSE. For practical applications in the field of robot control, however, this accuracy is generally adequate, considering that the logic test for the equality of real numbers can sensibly be carried out only within a small tolerance range.

Examples of permissible assignments to real variables:

```

REALNO1 = -13.653
REALNO2 = 10
REALNO3 = 34.56 E-12

```



If a REAL variable is assigned an INTEGER value, automatic type conversion to REAL is carried out. According to the above assignment, the variable REALNO2 therefore has the value 10.0!

BOOL

The boolean variables are used to describe logic states (e.g. input/output states). They can only have the value TRUE or FALSE:

```

STATE1 = TRUE
STATE2 = FALSE

```

CHAR

Character variables can represent exactly 1 character from the ASCII set of characters. In the assignment of an ASCII character to a CHAR variable, the assigned character must be placed between quotation marks (").

```
CHAR1 = "G"
```

```
CHAR2 = "?"
```

For information on storing entire character strings, see Section 2.2.4.

2.2.3 Arrays

The term "arrays" refers to the combination of objects of the same data type to form a data object; the individual components of an array can be addressed via indices. By means of the declaration

```
DECL INT OTTO[7]
```

Array index

You can store, for example, 7 different integers in the array `OTTO[]`. You can access each individual component of the array by specifying the associated index (the first index is always the number 1).

```
OTTO[1] = 5 ; The first element is assigned the number 5
OTTO[2] = 10 ; The second element is assigned the number 10
OTTO[3] = 15 ; The third element is assigned the number 15
OTTO[4] = 20 ; The fourth element is assigned the number 20
OTTO[5] = 25 ; The fifth element is assigned the number 25
OTTO[6] = 30 ; The sixth element is assigned the number 30
OTTO[7] = 35 ; The seventh element is assigned the number 35
```

It is helpful to imagine the array with the name `OTTO[]` as a rack with 7 compartments. In accordance with the above assignments, the compartments would then be filled as follows:

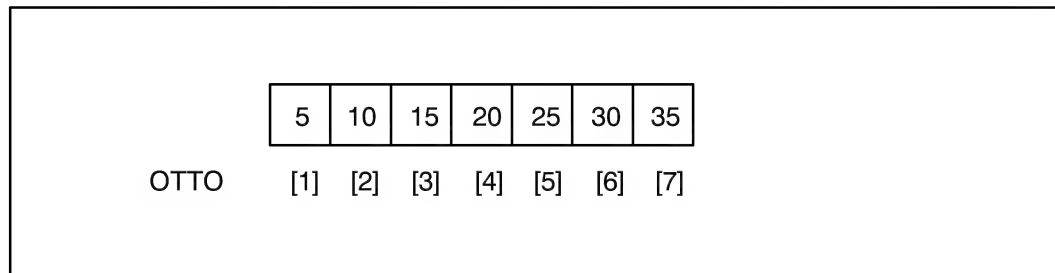


Fig. 2 Representation of a one-dimensional array

If all the elements of an array are now to be initialized with the same number, e.g. 0, you do not have to program each assignment explicitly but can "automate" the preassignment with the aid of a loop and a counting variable.

```
FOR I = 1 TO 7
    OTTO[I] = 0
ENDFOR
```



Further information can be found in the chapter **[Program execution control]**, section **[Loops]**.

In this case the counting variable is the integer variable `I`. It must be declared before being used as an integer.



- An array may be of any data type. The individual elements can thus in turn consist of composite data types (e.g. an array made up of arrays).
- Only integer data types are allowed for the index.
- Apart from constants and variables, arithmetic expressions are also allowed for the index (see Section 2.3.1).
- The index always starts at 1.

2D arrays

Besides the one-dimensional arrays already discussed, i.e. arrays with only one index, you can also use two- or three-dimensional arrays in KRL. By means of

```
DECL REAL MATRIX[ 7, 3 ]
```

you can declare a two-dimensional 5×4 array with $5 \times 4 = 20$ REAL elements. It is helpful to represent this array as a matrix with 5 columns and 4 rows. With the program sequence

```
I[ 3 ] = 0
FOR COLUMN = 1 TO 5
  FOR ROW = 1 TO 4
    I[ 3 ] = I[ 3 ] + 1
    MATRIX[ COLUMN, ROW ] = I[ 3 ]
  ENDFOR
ENDFOR
```

the elements of the matrix are assigned a value according to their sequence in the matrix. The following matrix assignment is thus obtained:

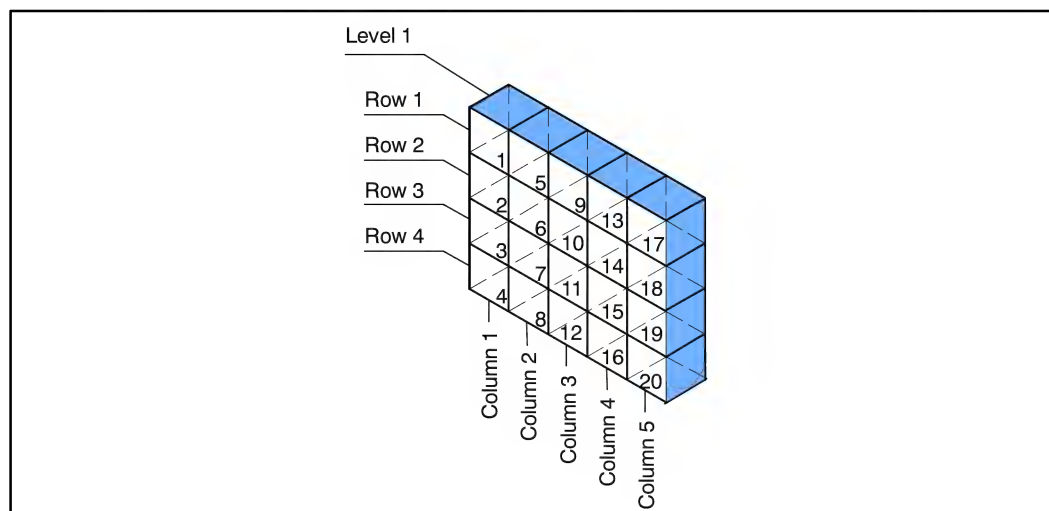


Fig. 3 Representation of a two-dimensional array

3D arrays

Three-dimensional arrays can be envisaged as several two-dimensional matrices one behind the other. The third dimension indicates, as it were, the level at which the matrix is located (see Fig. 4). A three-dimensional array is declared similarly to the one- or two-dimensional arrays, e.g.:

```
DECL BOOL ARRAY_3D[5,3,4]
```

The initialization sequence could then be as follows:

```
FOR LEVEL = 1 TO 3
  FOR COLUMN = 1 TO 5
    FOR ROW= 1 TO 4
      ARRAY_3D[LEVEL,COLUMN,ROW] = FALSE
    ENDFOR
  ENDFOR
ENDFOR
```

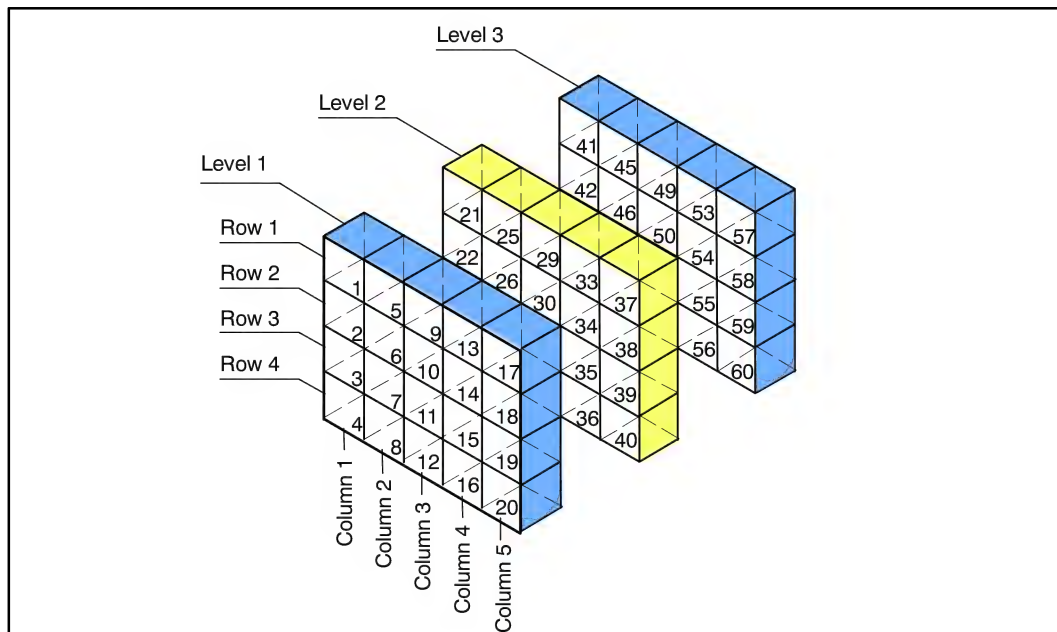


Fig. 4 Representation of a three-dimensional array

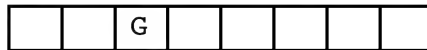
2.2.4 Character strings

Using the data type CHAR, you can only store individual characters, as described. For the purpose of using entire strings of characters, e.g. words, you simply define a one-dimensional array of type CHAR:

```
DECL CHAR NAME[ 8 ]
```

As usual, you can address each individual element of the array NAME[], e.g.:

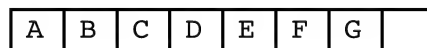
```
NAME[ 3 ] = "G"
```



However, you can also enter entire strings straight away:

```
NAME[ ] = "ABCDEFGG"
```

assigns to the first seven elements of the array NAME[] the letters A, B, C, D, E, F and G:



2.2.5 Structures

STRUC

If different data types are to be combined, the array is not suitable and the more general form of linkage must be used. Using the declaration statement STRUC, different data types which have been previously defined or are predefined data types are combined to form a new composite data type. In particular, other composites and arrays can also form part of a composite.

A typical example of the use of composites is the standard data type POS. It consists of 6 REAL values and 2 INT values and has been declared in the file \$OPERATE.SRC:

```
STRUC POS REAL X, Y, Z, A, B, C, INT S, T
```

Point
separator

If, for example, you now use a POSITION variable of the structure data type POS, you can assign values to the elements either individually with the aid of the point separator, e.g.:

```
POSITION.X = 34.4
POSITION.Y = -23.2
POSITION.Z = 100.0
POSITION.A = 90
POSITION.B = 29.5
POSITION.C = 3.5
POSITION.S = 2
POSITION.T = 6
```

Aggregate

or jointly by means of a so-called aggregate:

```
POSITION={X 34.4,Y -23.2,Z 100.0,A 90,B 29.5,C 3.5,S 2,T 6}
```



Further information can be found in the chapter **[Variables and declarations]**, section **[Declaration and initialization of data objects]**.

Aggregates are subject to the following conditions:



- The values of an aggregate can be simple constants or themselves aggregates.
- Not all components of the structure have to be specified in an aggregate.
- The components do not need to be specified in the order in which they have been defined.
- Each component may only be contained once in an aggregate.
- In the case of arrays consisting of structures, an aggregate defines the value of an individual array element.
- The name of the structure type can be specified at the beginning of an aggregate – separated by a colon.

The following assignments are thus also permissible for POS variables, for instance:

```
POSITION={B 100.0,X 29.5,T 6}
POSITION={A 54.6,B -125.64,C 245.6}
POSITION={POS: X 230,Y 0.0,Z 342.5}
```

In the case of POS, E6POS, AXIS, E6AXIS and FRAME structures missing components are not altered. In all other aggregates, non-existing components are set to invalid.

The procedure for creating your own structure variables will be explained with the aid of the following example:

In a subprogram for arc welding, the following information is to be transferred in a variable S_PARA:

REAL	V_WIRE	Wire speed
INT	CHARAC	Characteristic 0...100%
BOOL	ARC	with/without arc (for simulation)

The variable S_PARA must consist of 3 elements of a different data type. First of all, a new data type meeting these requirements must be created:

```
STRUC WELDTYPE REAL V_WIRE, INT CHARAC, BOOL ARC
```

A new data type with the designation WELDTYPE is thus created (WELDTYPE is not a variable!). WELDTYPE consists of the 3 components V_WIRE, CHARAC and ARC. You can now declare any variable of the new data type, e.g.:

```
DECL WELDTYPE S_PARA
```

You have thus created a variable S_PARA of the data type WELDTYPE. The individual elements can be addressed with the aid of the point separator or the aggregate – as already described.

```
S_PARA.V_WIRE = 10.2
S_PARA.CHARAC = 66
S_PARA.ARC = TRUE
```

or

```
S_PARA = {V_WIRE 10.2,CHARAC 50, ARC TRUE}
```



To make it easier to distinguish between self-defined data types of variables, the names of the new data types should end with ...TYPE.

Predefined Structures

The following structures are predefined in the file `$OPERATE.SRC`:

```
STRUC AXIS      REAL  A1,A2,A3,A4,A5,A6
STRUC E6AXIS    REAL  A1,A2,A3,A4,A5,A6,E1,E2,E3,E4,E5,E6
STRUC FRAME     REAL  X,Y,Z,A,B,C
STRUC POS       REAL  X,Y,Z,A,B,C, INT S,T
STRUC E6POS     REAL  X,Y,Z,A,B,C,E1,E2,E3,E4,E5,E6, INT S,T
```

The components **A1...A6** of the structure **AXIS** are angle values (rotational axes) or translation values (translational axes) for the axis-specific movement of robot axes 1...6.

Using the additional components **E1...E6** in the structure **E6AXIS**, external axes can be addressed.

In the structure **FRAME** you can define 3 position values in space (x, y and z) and 3 orientations in space (A, B and C). A point in space is thus unambiguously defined in terms of position and orientation.

As there are robots that can address one and the same point in space with several axis positions, the integer variables **S** and **T** in the structure **POS** are used to define an unambiguous axis position.



Further information can be found in the chapter **[Motion programming]**, section **[Motion commands]** Status (S) and Turn (T).

By means of the components **E1...E6** in the structure **E6POS**, external axes can again be addressed.

Geometric data types

The types **AXIS**, **E6AXIS**, **POS**, **E6POS** and **FRAME** are also called geometric data types because they provide the programmer with a simple means of describing geometric relations.



Further information can be found in the chapter **[Motion programming]**, section **[Application of the various coordinate systems]**.

2.2.6 Enumeration types

An enumeration data type consisting of a limited set of constants. The constants are freely selectable names and can be defined by the user. A variable of this data type (enumeration variable) can only take on the value of one of these constants.

This will be explained on the basis of the system variable `$MODE_OP`, in which the operating mode currently selected is stored. The modes **T1**, **T2**, **AUT** and **EX** are available for selection.

One could declare `$MODE_OP` as an integer variable, assign each mode a number and then store this number in `$MODE_OP`. That would not be very clear, however.

ENUM

A much more elegant solution is provided by the enumeration type. In the file `$OPERATE.SRC` an enumeration data type with the name `MODE_OP` has been generated:

```
ENUM MODE_OP  T1, T2, AUT, EX, INVALID
```

The command for declaring enumeration types is therefore called **ENUM**. Variables of the enumeration type `MODE_OP` can only have the values **T1**, **T2**, **AUT**, **EX** or **INVALID**. The variables are again declared using the keyword **DECL**:

```
DECL MODE_OP  $MODE_OP
```

sign

You can now allocate one of the four values of the data type `MODE_OP` to the enumeration variable `$MODE_OP` by means of a normal assignment. As a means of distinguishing them from simple constants, the self-defined enumeration constants are preceded by a “#” sign in initializations or queries, e.g.:

```
$MODE_OP = #T1
```

By means of `ENUM`, you can now generate any number of self-defined enumeration data types.



NOTES:

2.3 Data manipulation

For manipulating the various data objects, there are a host of operators and functions available, with the aid of which formulae can be established. The power of a robot programming language depends equally on the permissible data objects and their manipulation capabilities.

2.3.1 Operators

Operand

The term “operators” refers to the usual mathematical operators as opposed to functions such as $\text{SIN}(30)$, which supplies the sine of the angle 30° . In the operation $5+7$, 5 and 7 are therefore called operands and + the operator.

In each operation, the compiler checks the legitimacy of the operands. For example, $7 - 3$ is a legitimate operation as the subtraction of two integers, whereas $5 + \text{“A”}$ is an inadmissible operation as the addition of an integer and a character.

In many operations, such as $5 + 7.1$, i.e. the addition of integer and real values, type matching is carried out, the integer value 5 being converted to the real value 5.0. This topic is dealt with in greater detail in the discussion of the individual operators.

2.3.1.1 Arithmetic operators

Arithmetic operators concern the data types INTEGER and REAL. All 4 basic arithmetic operations are allowed in KRL (see Table 3).

Operator	Description
+	Addition or positive sign
-	Subtraction or negative sign
*	Multiplication
/	Division

Table 3 Arithmetic operators

The result of an arithmetic operation is only INT if both operands are of the data type INTEGER. If the result of an integer division is not an integer, it is cut off at the decimal point. If at least one of the two operands is REAL, the result too will be of the data type REAL (see Table 4).

Operands	INT	REAL
INT	INT	REAL
REAL	REAL	REAL

Table 4 Result of an arithmetic operation

The following program example is intended to illustrate this:



```

DEF ARITH()

;----- Declaration section -----
INT A,B,C
REAL K,L,M

;----- Initialization section -----
;All variables are invalid prior to initialization!
A = 2           ;A=2
B = 9.8         ;B=10
C = 7/4         ;C=1
K = 3.5         ;K=3.5
L = 0.1 E01     ;L=1.0
M = 3           ;M=3.0

;----- Main section -----
A = A * C       ;A=2
B = B - 'HB'    ;B=-1
C = C + K       ;C=5
K = K * 10      ;K=35.0
L = 10 / 4      ;L=2
L = 10 / 4.0    ;L=2.5
L = 10 / 4.     ;L=2.5
L = 10./ 4      ;L=2.5
C = 10./ 4.     ;C=3
M = (10/3) * M  ;M=9.0

END

```

2.3.1.2 Geometric operator

The geometric operator is symbolized by a colon ":" in KRL. It performs a frame linkage (logic operation) on operands of the data types **FRAME** and **POS**.

The linkage of two frames is the usual transformation of coordinate systems. The linkage of a **FRAME** structure and a **POS** structure therefore only affects the frame within the **POS** structure. The components **S** and **T** remain unaffected by the transformation and therefore do not have to be assigned a value. The values **X**, **Y**, **Z**, **A**, **B** and **C** must, however, always be assigned a value in both **POS** operands and **FRAME** operands.

Frame
linkage

A frame operation is evaluated from left to right. The result always has the data type of the operand on the far right (see Table 5).

Left operand (reference CS)	Operator	Right operand (target CS)	Result
POS	:	POS	POS
POS	:	FRAME	FRAME
FRAME	:	POS	POS
FRAME	:	FRAME	FRAME

Table 5 Data type combinations with the geometric operator



If the left-hand operand has the data type POS, type matching takes place. The position specified by the POS structure is transformed into a frame. That means the system determines the tool frame for this position.

A simple example will be used in order to explain the mode of functioning of the geometric operator (see Fig. 5):

In a room there is a table. The ROOM coordinate system is defined as a fixed coordinate system with its origin at the front left corner of the room.

The table is located parallel to the walls of the room. The front left corner of the table is located exactly 600 mm from the front wall and 450 mm from the left-hand wall of the room. The table is 800 mm high.

On the table is a cuboidal workpiece. The WORKPIECE coordinate system has its origin at one corner of the workpiece, as shown in Fig. 35. To allow the part to be optimally handled in later operation, the Z-axis of the WORKPIECE coordinate system points downwards. The workpiece is rotated by 40° in relation to the Z-axis of the TABLE coordinate system. The position of the WORKPIECE coordinate system with reference to the TABLE coordinate system is $X = 80$ mm, $Y = 110$ mm and $Z = 55$ mm.

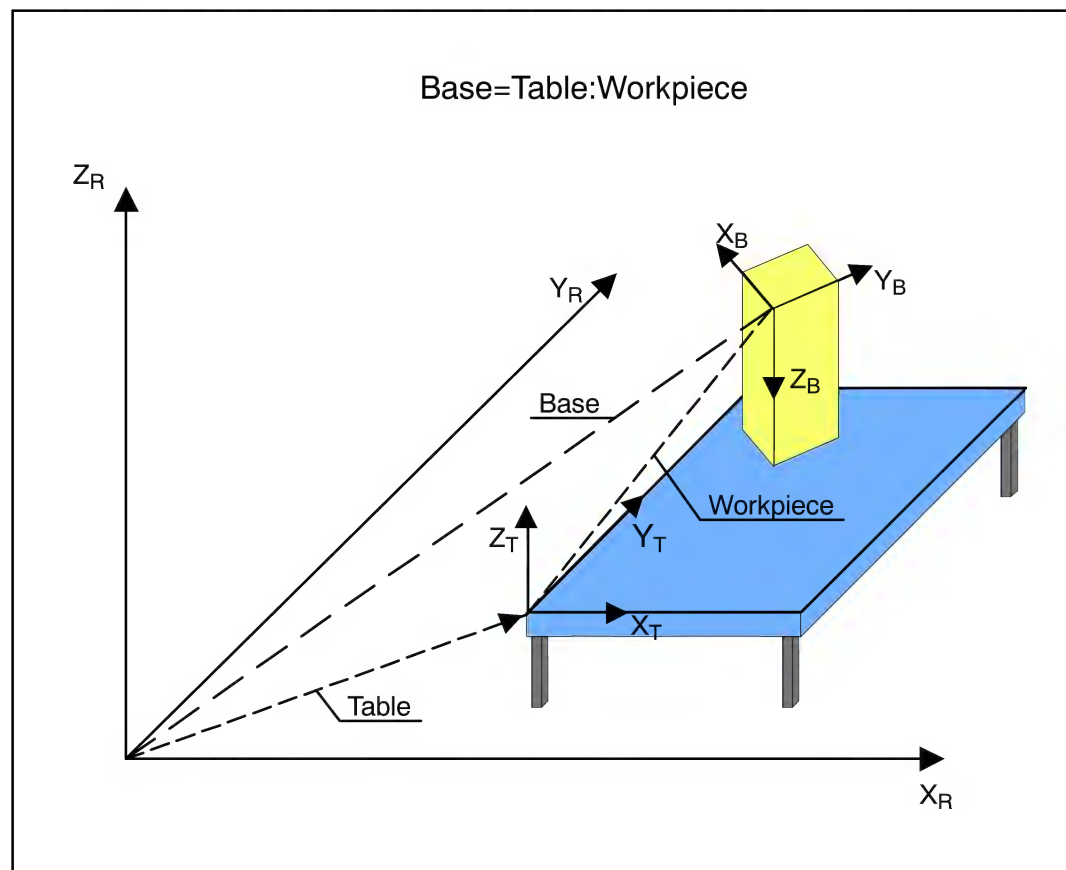


Fig. 5 Mode of functioning of the geometric operator

The task is now to define the WORKPIECE coordinate system in relation to the ROOM coordinate system. For this purpose, the following frame variables must first be defined:

```
FRAME TABLE, WORKPIECE, BASE
```

The ROOM coordinate system is already defined specifically to the system. The TABLE and WORKPIECE coordinate systems are now initialized in accordance with the given constraints.

```
TABLE = {X 450,Y 600,Z 800,A 0,B 0,C 0}
```

```
WORKPIECE = {X 80,Y 110,Z 55,A -40,B 180,C 0}
```


The `WORKPIECE` coordinate system in relation to the `ROOM` coordinate system is now obtained with the aid of the geometric operator as

```
BASE = TABLE:WORKPIECE
```

In our case, `BASE` is then defined as follows:

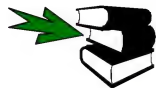
```
BASE = {X 530,Y 710,Z 855,A 140,B 0,C -180}
```

Another possibility would be:

```
BASE = {X 530,Y 710,Z 855,A -40,B 180,C 0}
```



Only in this specific case do the components of `BASE` result from the addition of the components of `TABLE` and `WORKPIECE`. This is due to the fact that the `TABLE` coordinate system is not rotated in relation to the `ROOM` coordinate system. In general, though, simple addition of the components is not possible! Frame linking is not commutative either, meaning that if the reference frame and the target frame are interchanged, the result too will normally change!



Further information can be found in the chapter **[Motion programming]**, section **[Application of the various coordinate systems]**.

Another example illustrating application of the geometric operator: Various coordinate systems and linkages of coordinate systems are addressed in this example. In order to illustrate changes in orientation, the tool center point is moved in each coordinate system first a short distance in the X direction, then in the Y direction and finally in the Z direction.



```

DEF  GEOM_OP ( );

----- Declaration section -----
EXT  BAS (BAS_COMMAND :IN,REAL :IN )
DECL AXIS HOME           ;Variable HOME of type AXIS
DECL FRAME MYBASE[2]     ;Array of type FRAME;

----- Initialization -----
BAS (#INITMOV,0 ) ;Initialization of velocities,
                  ;accelerations, $BASE, $TOOL, etc.
HOME={AXIS: A1 0,A2 -90,A3 90,A4 0,A5 30,A6 0}; Set base
coordinate system
$BASE={X 1000,Y 0,Z 1000,A 0,B 0,C 0}
REF_POS_X={X 100,Y 0,Z 0,A 0,B 0,C 0}           ;Reference
                                                position.
REF_POS_Y={X 100,Y 100,Z 0,A 0,B 0,C 0}
REF_POS_Z={X 100,Y 100,Z 100,A 0,B 0,C 0}       ;Define own
                                                coordinate systems
MYBASE[1]={X 200,Y 100,Z 0,A 0,B 0,C 180}
MYBASE[2]={X 0,Y 200,Z 250,A 0,B 90,C 0};

----- Main section -----
PTP  HOME ; Motion in relation to $BASE coordinate system
PTP  $Base ;Direct positioning to origin of $BASE-CS
WAIT SEC 2 ;Wait 2 seconds
PTP  REF_POS_X ;Move 100 mm in x direction
PTP  REF_POS_Y ;Move 100 mm in y direction
PTP  REF_POS_Z ;Move 100 mm in z direction; Motion in
relation to $BASE-CS offset by MYBASE[1]
PTP  MYBASE[1]
WAIT SEC 2
PTP  MYBASE[1]:REF_POS_X
PTP  MYBASE[1]:REF_POS_Y
PTP  MYBASE[1]:REF_POS_Z; Motion in relation to $BASE-CS offset
by MYBASE[2]
PTP  MYBASE[2]
WAIT SEC 2
PTP  MYBASE[2]:REF_POS_X
PTP  MYBASE[2]:REF_POS_Y
PTP  MYBASE[2]:REF_POS_Z; Motion in relation to $BASE-CS offset
by MYBASE[1]:MYBASE[2]
PTP  MYBASE[1]:MYBASE[2]
WAIT SEC 2
PTP  MYBASE[1]:MYBASE[2]:REF_POS_X
PTP  MYBASE[1]:MYBASE[2]:REF_POS_Y
PTP  MYBASE[1]:MYBASE[2]:REF_POS_Z; Motion in relation to
$BASE-CS offset by MYBASE[2]:MYBASE[1]
PTP  MYBASE[2]:MYBASE[1]
WAIT SEC 2
PTP  MYBASE[2]:MYBASE[1]:REF_POS_X
PTP  MYBASE[2]:MYBASE[1]:REF_POS_Y
PTP  MYBASE[2]:MYBASE[1]:REF_POS_Z
PTP  HOME
END

```

2.3.1.3 Relational operators

Using the relational operators listed in Table 6, it is possible to form logical expressions. The result of a comparison is therefore always of the data type `BOOL`, since a comparison can only ever be `TRUE` or `FALSE`.

Operator	Description	Permissible data types
<code>==</code>	equal to	<code>INT</code> , <code>REAL</code> , <code>CHAR</code> , <code>ENUM</code> , <code>BOOL</code>
<code><></code>	not equal to	<code>INT</code> , <code>REAL</code> , <code>CHAR</code> , <code>ENUM</code> , <code>BOOL</code>
<code>></code>	greater than	<code>INT</code> , <code>REAL</code> , <code>CHAR</code> , <code>ENUM</code>
<code><</code>	less than	<code>INT</code> , <code>REAL</code> , <code>CHAR</code> , <code>ENUM</code>
<code>>=</code>	greater than or equal to	<code>INT</code> , <code>REAL</code> , <code>CHAR</code> , <code>ENUM</code>
<code><=</code>	less than or equal to	<code>INT</code> , <code>REAL</code> , <code>CHAR</code> , <code>ENUM</code>

Table 6 Relational operators

Comparisons can be used in program execution instructions (see Section 5), and the result of a comparison can be assigned to a boolean variable.

The test for equality or inequality is of only limited use with real numbers since algebraically identical formulae can supply unequal values due to rounding errors in the calculation of the values to be compared (see 2.2.2).



- Operand combinations of `INT`, `REAL` and `CHAR` are possible.
- An `ENUM` type may only be compared with the same `ENUM` type.
- A `BOOL` type may only be compared with a `BOOL` type.

The comparison of numeric values (`INT`, `REAL`) and character values (`CHAR`) is possible because each ASCII character is assigned an ASCII code. This code is a number defining the order of the characters in the character set.

In their declaration, the individual constants of an enumeration type are numbered in the order of their occurrence. The relational operators refer to these numbers.

Both simple and multiple comparisons are permitted. Some examples to illustrate this:

```

...
BOOL A,B
...
B = 10 < 3                               ;B=FALSE
A = 10/3 == 3                             ;A=TRUE
B = ((B == A) <> (10.00001 >= 10)) == TRUE ;B=TRUE
A = "F" < "Z"                             ;A=TRUE
...

```

2.3.1.4 Logical operators

These operators are used for performing logical operations on boolean variables, constants and simple logical expressions, as are formed with the aid of relational operators. For example, the expression

$(A > 5) \text{ AND } (A < 12)$

has the value TRUE only if A lies in the range between 5 and 12. Such expressions are frequently used in instructions serving the purpose of checking program execution (see Section 5). The logical operators are listed in Table 7.

Operator	Operand number	Description
NOT	1	Inversion
AND	2	Logical AND
OR	2	Logical OR
EXOR	2	Exclusive OR

Table 7 Logical operators

The operands of a logical operation must be of type BOOL, and the result too is always of type BOOL. The possible results of the various logical operations are shown in Table 8 as a function of the value of the operands.

Operation		NOT A	A AND B	A OR B	A EXOR B
A = TRUE	B = TRUE	FALSE	TRUE	TRUE	FALSE
A = TRUE	B = FALSE	FALSE	FALSE	TRUE	TRUE
A = FALSE	B = TRUE	TRUE	FALSE	TRUE	TRUE
A = FALSE	B = FALSE	TRUE	FALSE	FALSE	FALSE

Table 8 Truth table for logical operations

Some examples of logical operations:

```

...
BOOL A,B,C
...
A = TRUE                                ;A=TRUE
B = NOT A                              ;B=FALSE
C = (A AND B) OR NOT (B EXOR NOT A)    ;C=TRUE
A = NOT NOT C                          ;A=TRUE
...

```

2.3.1.5 Bit operators

Using the bit operators (see Table 9), whole numbers can be combined by performing logical operations on the individual bits of the numbers. The bit operators combine individual bits just as the logical operators combine two boolean values if the bit value 1 is regarded as `TRUE` and the value 0 as `FALSE`.

Bit-by-bit `AND`ing of the numbers 5 and 12 thus produces the number 4, for example, bit-by-bit `OR`ing the number 13 and bit-by-bit exclusive `OR`ing the number 9:

	0	1	0	1	= 5
	1	1	0	0	= 12
AND	0	1	0	0	= 4
OR	1	1	0	1	= 13
EXOR	1	0	0	1	= 9

Bit-by-bit inversion does not simply involve all the bits being inverted. Instead, when `B_NOT` is used, 1 is added to the operand and the sign is changed, e.g.:

```
B_NOT 10 = -11
B_NOT -10 = 9
```

Bit operators are used, for example, to combine digital I/O signals (see 6.3).

Operator	Operand number	Description
B_NOT	1	Bit-by-bit inversion
B_AND	2	Bit-by-bit <code>AND</code> ing
B_OR	2	Bit-by-bit <code>OR</code> ing
B_EXOR	2	Bit-by-bit exclusive <code>OR</code> ing

Table 9 Logical bit operators



As ASCII characters can also be addressed via the integer ASCII code, the data type of the operands may also be `CHAR` besides `INT`. The result is always of type `INT`.

Examples illustrating the use of bit operators:

```
...
INT A
...
A = 10 B_AND 9           ;A=8
A = 10 B_OR 9            ;A=11
A = 10 B_EXOR 9          ;A=3
A = B_NOT 197            ;A=-198
A = B_NOT 'HC5'          ;A=-198
A = B_NOT 'B11000101'    ;A=-198
A = B_NOT "E"            ;A=-70
...
```

Let us assume you have defined an 8-bit digital output. You can address the output via the INTEGER variable DIG. To set bits 0, 2, 3 and 7, you can now simply program

Setting bits DIG = 'B10001101' B_OR DIG

All the other bits remain unaffected, regardless of their value.

If you want to mask out bits 1, 2 and 6, program

Masking out bits DIG = 'B10111001' B_AND DIG

All the other bits remain unaltered.

You can just as easily use the bit operators to check whether individual bits of the output are set. The expression

Filtering out bits ('B10000001' B_AND DIG) == 'B10000001'

becomes TRUE if bits 0 and 7 are set, otherwise it is FALSE.

If you only want to test whether at least one of the two bits 0 and 7 is set, the result of the bit-by-bit ANDing merely has to be greater than zero:

('B10000001' B_AND DIG) > 0

2.3.1.6 Priority of operators

If you use complex expressions with several operators, you must take into account the different priorities of the individual operators (see Table 10), as the various expressions are executed in the order of their priorities.

Priority

Priority	Operator
1	NOT B_NOT
2	* /
3	+ -
4	AND B_AND
5	EXOR B_EXOR
6	OR B_OR
7	== <> < > >= <=

Table 10 Priority of operators

General rules:

- Bracketed expressions are processed first.
- Non-bracketed expressions are evaluated in the order of their priority.
- Logic operations with operators of the same priority are executed from left to right.

Examples:

```

...
INT A,B
BOOL E,F
...
A = 4
B = 7
E = TRUE
F = FALSE
...
E = NOT E OR F AND NOT (-3 + A * 2 > B) ;E=FALSE
A = 4 + 5 * 3 - B_NOT B / 2 ;A=23
B = 7 B_EXOR 3 B_OR 4 B_EXOR 3 B_AND 5 ;B=5
F = TRUE == (5 >= B) AND NOT F ;F=TRUE
...

```


2.3.2 Standard functions

For calculating certain mathematical problems, a number of standard functions are predefined in KRL (see Table 11). They can be used directly without further declaration.

Description	Function	Data type of argument	Range of values of argument	Data type of function	Range of values of result
Absolute value	ABS (X)	REAL	$-\infty \dots +\infty$	REAL	$0 \dots +\infty$
Square root	SQRT (X)	REAL	$0 \dots +\infty$	REAL	$0 \dots +\infty$
Sine	SIN (X)	REAL	$-\infty \dots +\infty$	REAL	$-1 \dots +1$
Cosine	COS (X)	REAL	$-\infty \dots +\infty$	REAL	$-1 \dots +1$
Tangent	TAN (X)	REAL	$-\infty \dots +\infty^*$	REAL	$-\infty \dots +\infty$
Arc cosine	ACOS (X)	REAL	$-1 \dots +1$	REAL	$0^\circ \dots 180^\circ$
Arc tangent	ATAN2 (Y, X)	REAL	$-\infty \dots +\infty$	REAL	$-90^\circ \dots +90^\circ$
* no odd multiple of 90° , i.e. $x \neq (2k-1) \cdot 90^\circ$, $k \in \mathbb{N}$					

Table 11 Mathematical standard functions

Absolute value

The function **ABS (X)** calculates the absolute value x, e.g.:

```
B = -3.4
A = 5 * ABS(B) ;A=17.0
```

Square root

SQRT (X) calculates the square root of the number x, e.g.:

```
A = SQRT(16.0801) ;A=4.01
```

Sine
Cosine
Tangent

The trigonometric functions **SIN (X)**, **COS (X)** and **TAN (X)** calculate the sine, cosine or tangent of the angle x, e.g.:

```
A = SIN(30) ;A=0.5
B = 2 * COS(45) ;B=1.41421356
C = TAN(45) ;C=1.0
```

The tangent of $\pm 90^\circ$ and odd multiples of $\pm 90^\circ$ ($\pm 270^\circ$, $\pm 450^\circ$, $\pm 630^\circ$...) is infinite. An attempt to calculate one of these values therefore generates an error message.

Arc cosine

ACOS (X) is the inverse function of COS(X):

```
A = COS(60) ;A=0.5
B = ACOS(A) ;B=60
```

Arc sine

There is no standard function predefined for arc sine, the inverse function of SIN (X). However, this can be very easily calculated on the basis of the relationship $\text{SIN}(X) = \text{COS}(90^\circ - X)$:

```
A = SIN(60) ;A=0.8660254
B = 90 - ACOS(A) ;B=60
```

Arc tangent

The tangent of an angle is defined as the opposite side (y) divided by the adjacent side (x) of a right triangle. Knowing the length of the two legs of the triangle, it is therefore possible to calculate the angle between the adjacent side and the hypotenuse by means of the arc tangent.

If we now consider a full circle, the sign of the components x and y is of decisive importance. If we were only to consider the quotient, it would only be possible to calculate angles between 0° and 180° by means of the arc tangent. This is also the case with all customary pocket calculators: The arc tangent of positive values gives an angle between 0° and 90° , the arc tangent of negative values an angle between 90° and 180° .

By the explicit specification of x and y , the quadrant in which the angle is located is unambiguously defined by their signs (see Fig. 6). You can therefore also calculate angles in quadrants III and IV. For calculating the arc tangent in the function **ATAN2** (y, x), these two specifications are therefore also necessary, e.g.:

```
A = ATAN2 ( 0.5, 0.5 )           ;A=45
B = ATAN2 ( 0.5, -0.5 )         ;B=135
C = ATAN2 (-0.5, -0.5 )         ;C=225
D = ATAN2 (-0.5, 0.5 )         ;D=315
```

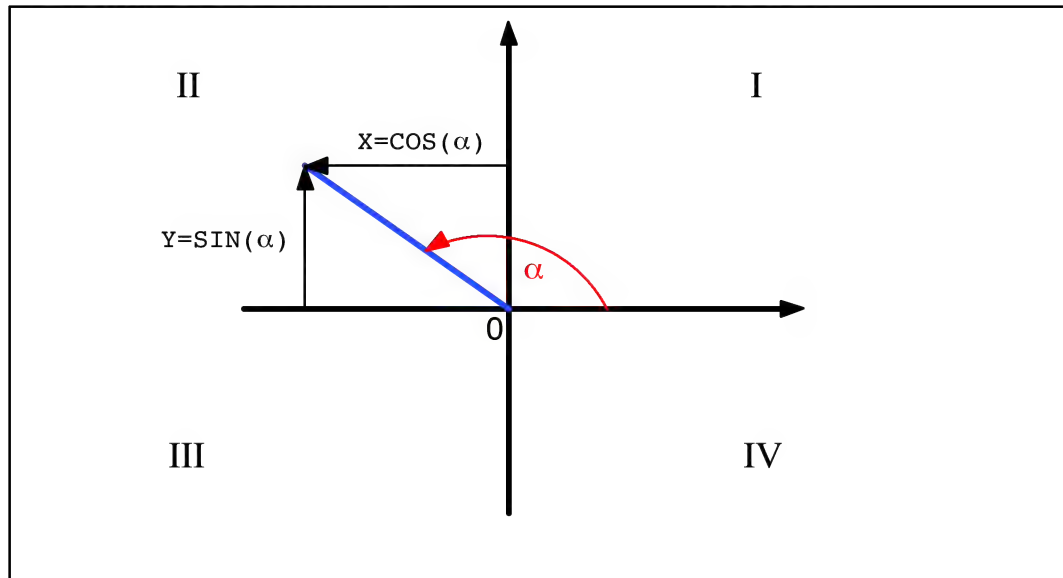


Fig. 6 Use of x and y in the function **ATAN2** (y, x)



Further information can be found in the chapter **[Subprograms and functions]**.



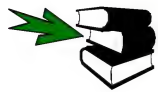
NOTES:

2.4 System variables and system files

An important precondition for processing complex robotic applications is a freely and easily programmable controller.

To meet this requirement, the functionality of the robot controller must be programmable in the robot language. The full functionality of a robot controller can only be utilized if the control parameters can be completely and yet easily integrated into a robot program. This is optimally solved in the KR C... by means of the concept of predefined system variables and files.

Examples of predefined variables are \$POS_ACT (current robot position), \$BASE (base coordinate system) or \$VEL.CP (CP velocity). A detailed description and a list of all predefined variables can be found in the training manual.



A list of all the predefined variables may be found in the separate documentation **[System variables]**.

System variables are completely integrated into the variables concept of KRL. They possess a corresponding data type and can be read and written by you like any other variable in the program provided there are no restrictions due to the type of data. The current robot position, for example, cannot be written but only read. Restrictions of this nature are checked by the controller.

As far as is permitted by the safety concept, you even have write access to system data. This creates a wide range of diagnostic capabilities since a large number of system data can be loaded or influenced from the KCP or programming system.

Examples of useful system variables with write access are \$TIMER[] and \$FLAG[].

Timers

The 16 timer variables \$TIMER[1]...\$TIMER[16] serve the purpose of measuring time sequences and can thus be used as a "stopwatch". A timing process is started and stopped by means of the system variables \$TIMER_STOP[1]...\$TIMER_STOP[16]:

```
$TIMER_STOP[ 4 ] = FALSE
```

starts timer 4, for example.

```
$TIMER_STOP[ 4 ] = TRUE
```

stops timer 4 again. The timer variable concerned can be reset at any time using a normal value assignment, e.g.:

```
$TIMER[ 4 ] = 0
```

If the value of a timer variable changes from minus to plus, a corresponding flag is set to TRUE (timer-out condition), e.g.:

```
$TIMER_FLAG[ 4 ] = TRUE
```

When the controller is booted, all the timer variables are preset to 0, the flags \$TIMER_FLAG[1]...\$TIMER_FLAG[16] to FALSE and the variables \$TIMER_STOP[1]...\$TIMER_STOP[16] to TRUE.

The unit of the timer variables is milliseconds (ms). \$TIMER[1]...\$TIMER[16] and \$TIMER_FLAG[1]...\$TIMER_FLAG[16] are updated in a 12 ms cycle.

Flags The 1024 flags `$FLAG[1]...$FLAG[1024]` are used as global markers. These boolean variables are preset to `FALSE`. You can view the current values of the markers on the user interface at any time by means of the “Monitor” menu item.

Cyclical flags There are also 32 cyclical flags `$CYCFLAG[1]...$CYCFLAG[32]` available in the KR C... . They are all preset to `FALSE` after the controller has booted.

The flags are cyclically active at the robot level only. Cyclical flags are permissible in a submit file, but they are not cyclically evaluated.

Cyclical flags can also be defined and activated in subprograms, functions and interrupt subprograms.

`$CYCFLAG[1]...$CYCFLAG[32]` have the data type `BOOL`. Any boolean expression can be used in an assignment to a cyclical flag.

The following are allowed:

- Boolean system variables
- Boolean variables which have been declared and initialized in a data list.

Not allowed on the other hand are

- functions that return a boolean value.

The statement

```
$CYCFLAG[10] = $IN[2] AND $IN[13]
```

has the effect that the boolean expression “`$IN[2] AND $IN[13]`” is cyclically evaluated, for example. This means that as soon as input 2 or input 13 changes, `$CYCFLAG[10]` changes too, regardless of the location of the program pointer after the above expression has been executed.

All the cyclical flags defined remain valid until a module is deselected or block selection is carried out by means of reset. All the cyclical flags remain active when the end of the program is reached.



Further information can be found in the chapter **[Interrupt handling]**, section **[Use of cyclical flags]**.

\$ sign

The names of the predefined variables have generally been chosen to allow them to be easily remembered. They all begin with a \$ sign and then consist of a meaningful English abbreviation. As they are treated like normal variables, you do not have to remember any unusual commands or rare options.

To avoid any risk of confusion, you should not declare any variables yourself which begin with a \$ sign.

Some of the predefined variables refer to the overall KR C... controller (e.g. `$ALARM_STOP` for defining the output for the Emergency Stop signal to the PLC). Others, however, are of relevance to the robot only (e.g. `$BASE` for the base coordinate system).

The robot drive on which the control-relevant data are stored in the directory “Steu” and the robot-relevant data are stored in the directory “R1” is displayed in the KUKA GUI.

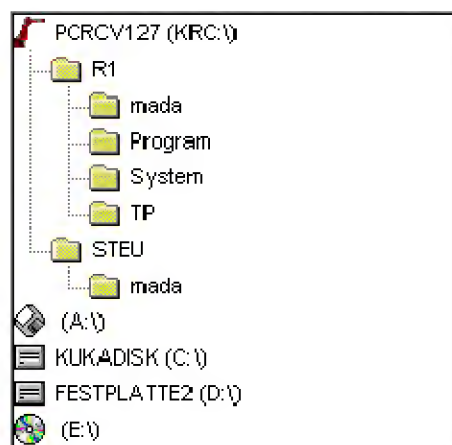


Fig. 7 Various levels on the KUKA graphic user interface

When programming the KR C..., you can create program files and data lists. Located in the program files are data definitions and executable instructions, while the data lists only contain data definitions and possibly initializations.



Further information can be found in the chapter **[Data lists]**.

In addition to the data lists that you create during programming, there are also data lists in the KR C... that are defined by KUKA and supplied with the control software. These data lists are called predefined data lists and mainly contain the predefined variables.

You can neither delete nor generate the predefined data lists yourself. They are generated when the software is installed and are then always available. Like the names of the predefined data, the names of the predefined data lists also begin with a \$ sign.

The following predefined data lists exist in the KR C...:

- **\$MACHINE.DAT**
is a predefined data list with exclusively predefined system variables. The machine data serve the purpose of adapting the controller to the connected robot (kinematic information, control parameters, etc.). There is a \$MACHINE.DAT in both the control system and the robot system. You cannot create new variables or delete existing ones.

Examples:

\$ALARM_STOP

Signal for Emergency Stop (control-specific)

\$NUM_AX

Number of robot axes (robot-specific)

- **\$CUSTOM.DAT**
is a data list that only exists in the control system. It contains data with which you can configure or parameterize certain control functions. The programmer is only able to alter the values of the predefined variables. You cannot create new variables or delete existing ones.

Examples:

\$PSER_1

Protocol parameters of serial interface 1

\$IBUS_ON

Activation of alternative Interbus groups

▪ **\$CONFIG.DAT**

is a data list predefined by KUKA which does not contain any system variables, however. There is a \$CONFIG.DAT available at both the control level and the robot level. Variables, structures, channels and signals can be defined in it which are valid over a long time and are of general significance for a lot of programs.

The data list is divided into the following blocks:

- BAS
- AUTOEXT
- GRIPPER
- PERCEPT
- SPOT
- A10
- A50
- A20
- TOUCHSENSE
- USER

Global declarations by the user should always be entered in the USER block since only here will the declarations be transferred in a later software upgrade.

▪ **\$ROBCOR.DAT**

The file \$ROBCOR.DAT contains robot-specific data for the dynamic model of the robot. These data are required for path planning. You cannot create new variables or delete existing ones in this file either.

Table 12 provides a summary of the predefined data lists.

Data list	System		Value assignment	
	Control	Robot	at	by
\$MACHINE.DAT	✓	✓	commissioning	KUKA/user
\$CUSTOM.DAT	✓		commissioning	user/KUKA
\$CONFIG.DAT	✓	✓	cell installation or conversion	user/KUKA
\$ROBCOR.DAT		✓	delivery	KUKA

Table 12 Predefined data lists in the KR C...

3 Motion programming

Motion instructions

One of the most important tasks of the robot controller is moving the robot. The programmer controls the movements of the industrial robot by means of special motion commands. These are also the main features which distinguish robot languages from conventional computer programming languages such as C or Pascal.

Depending on the type of control, these motion instructions can be subdivided into commands for simple point-to-point motions and commands for continuous-path movements. Whereas, with continuous-path movements, the end effector (e.g. gripper or tool) describes a precise, geometrically defined path in space (straight line or arc), the motion path in point-to-point movements is dependent on the robot's kinematic system and cannot, therefore, be accurately predicted. Common to both these types of motion is that programming takes place from the current position to a new position. For this reason, a motion instruction generally only requires the specification of the end position (exception: circular motions, see 3.3.4).

Position coordinates can be specified either as text, by entering numeric values, or by moving the robot to them and saving the actual values (teaching). The possibility exists, in each case, of relating the entries to various coordinate systems.

Further motion properties, such as velocity and acceleration, and orientation control, can be set using system variables. The approximation of auxiliary points is initiated with the aid of optional parameters in the motion instruction. In order to carry out approximation, a computer advance run must be set.

3.1 Application of the various coordinate systems

Coordinate system

Various coordinate systems are used in order to be able to specify the position or orientation of a point in space. A fundamental distinction can be made between joint (axis-specific) and Cartesian coordinate systems:

Axis-specific

In the **joint coordinate system**, the linear shifts (for translational axes) or the rotational values (for rotational axes) are specified for each robot axis. In the case of a jointed-arm robot with 6 axes, all 6 robot joint angles must therefore be entered in order to define the position and orientation unambiguously (see Fig. 8).

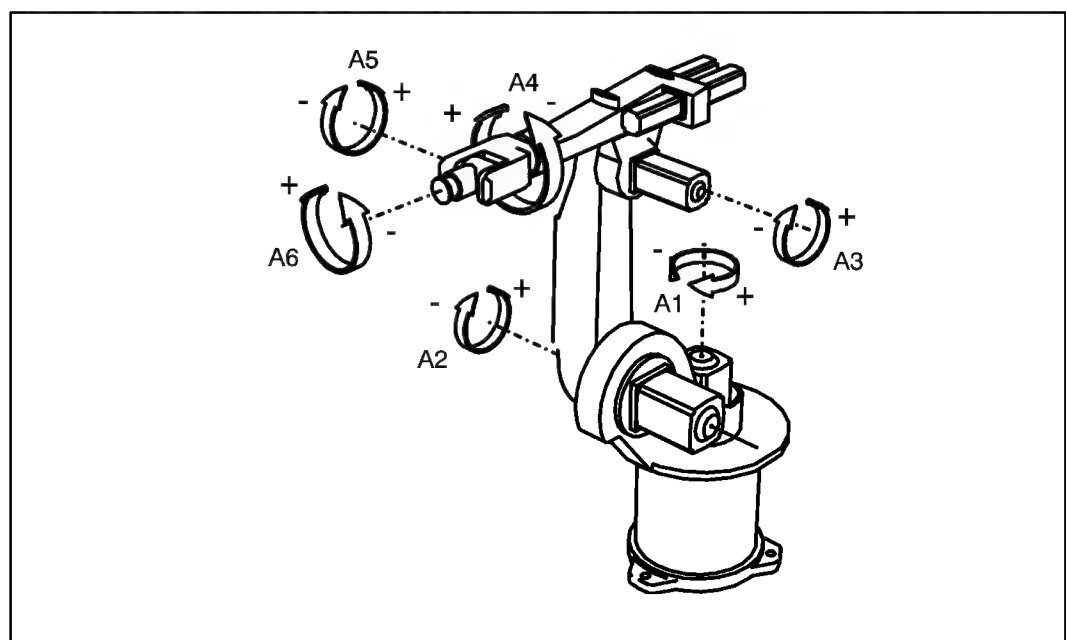


Fig. 8 Joint coordinate system for a jointed-arm robot with 6 axes

An axis-specific position is defined in the KR C... using the predefined structure type **AXIS**, the components of which signify angles or lengths depending on the type of axis.



Further information can be found in the chapter **[Variables and declarations]**, section **[Structures]**.

Coordinate transformation

Axis-specific positions can only be used in conjunction with PTP motion commands. If a CP motion is programmed with an axis-specific robot position, a fault situation will arise.

Since the programmer is human, and therefore thinks in Cartesian coordinates, programming in the joint coordinate system is usually highly impractical. For this reason, the controller offers several Cartesian coordinate systems for programming purposes, the coordinates of which are then automatically converted for use in the joint coordinate system before the motion is executed (see Fig. 9).

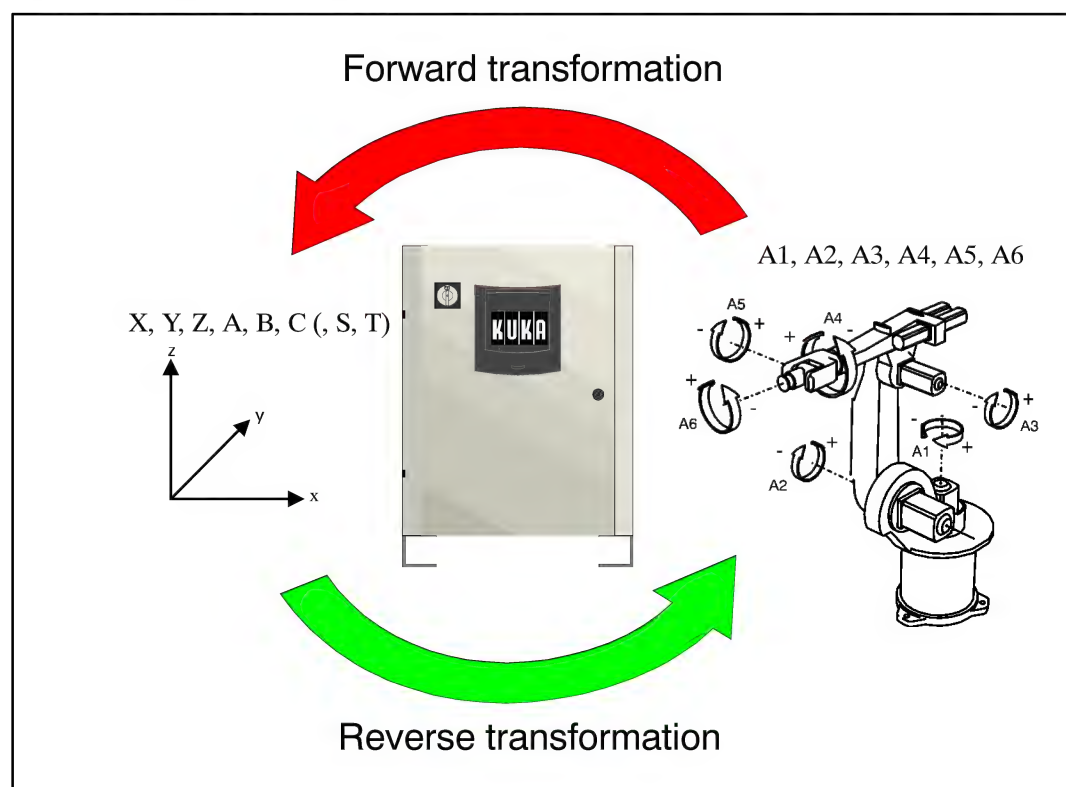


Fig. 9 Coordinate transformation

Cartesian

In a **Cartesian coordinate system**, the X, Y and Z coordinate axes lie perpendicular to one another and constitute, in this order, a rectangular system.

The position of a point in space in the Cartesian coordinate system is unambiguously determined by specifying the X, Y and Z coordinates. These are derived from the translational distance of each coordinate value from the coordinate origin (see Fig. 10).

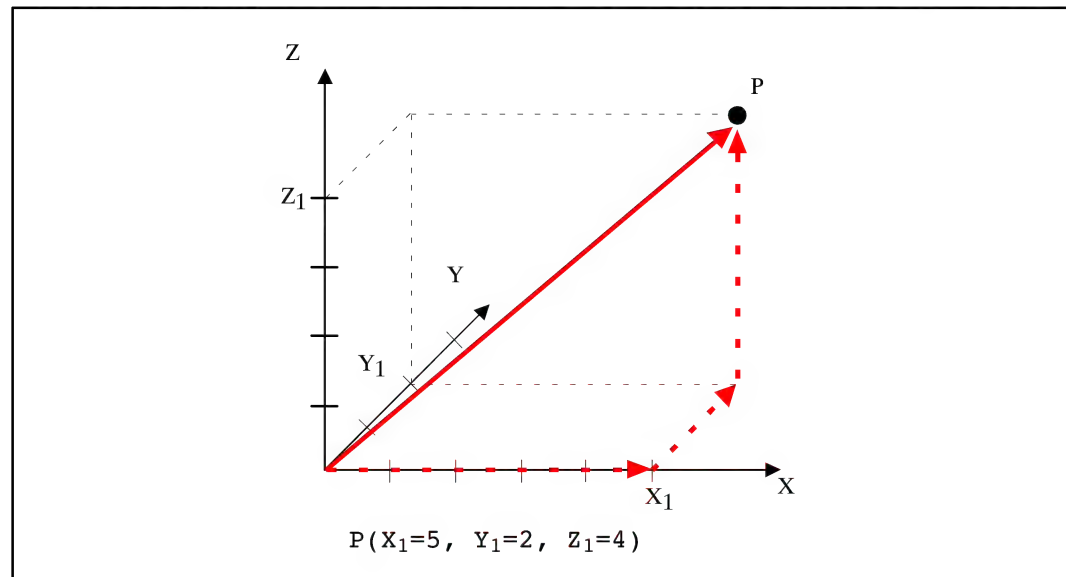


Fig. 10 Translational description of the position of a point

In order to be able to position the robot to each point in space, whatever orientation it has, three rotational specifications are required in addition to the three translational values:

The angles designated A, B and C in the KR C... describe rotations about the coordinate axes Z, Y and X. The order of the rotations must be retained.

1. Rotation through angle A about the Z axis
2. Rotation through angle B about the new Y axis
3. Rotation through angle C about the new X axis

This rotation sequence corresponds to the well-known roll-pitch-yaw angles in the field of aviation. Angle C corresponds in this case to the roll, angle B to the pitch and angle A to the yaw (see Fig. 11).

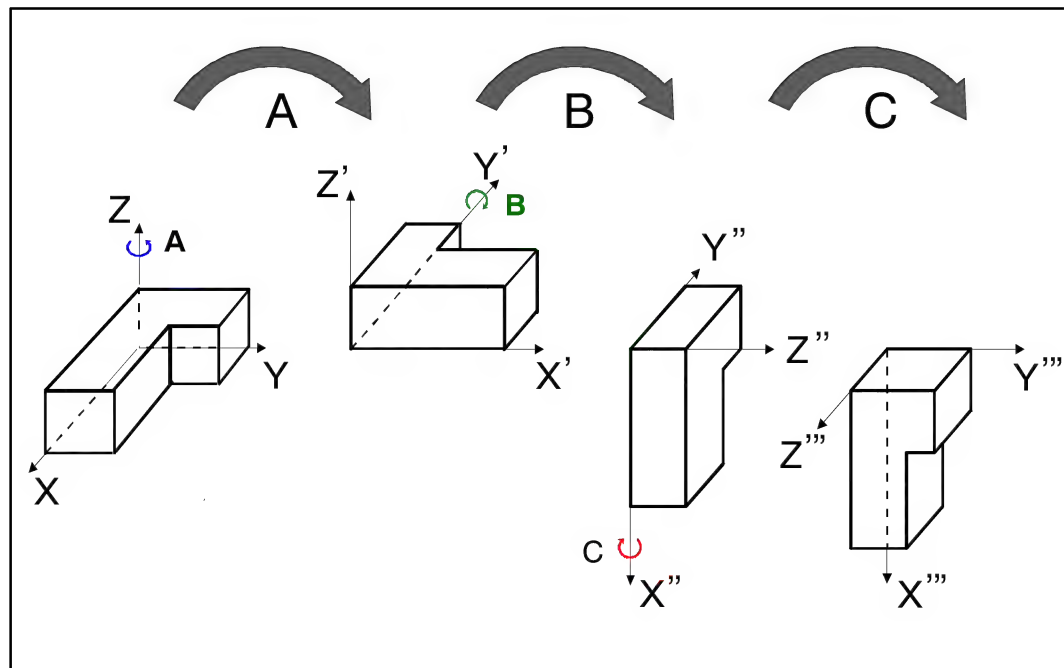
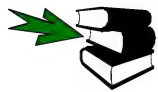


Fig. 11 Rotational description of the orientation of a point

Translations X, Y and Z, along with the rotations A, B and C, allow the unambiguous definition of the position and orientation of a point in space. The KR C... does this using the predefined structure `FRAME`.



Further information can be found in the chapter **[Variables and declarations]**, section **[Structures]**.

With continuous-path motions, the specification of `FRAME` coordinates is always sufficient and unambiguous. In the case of PTP motions, however, and with certain robot kinematic systems (e.g. jointed arm with 6 axes), a point in space (position and orientation defined) can be reached with several different axis positions. This ambiguity can be rectified by means of the two additional entries S and T. The extension of a frame to include S and T is catered for in the KR C... by the structure `POS`.



Further information can be found in the chapter **[Variables and declarations]**, section **[Structures]** and section **[Motion commands]**.

The following Cartesian coordinate systems are predefined in the KR C... (Table 13 and Fig. 12):

Coordinate system	System variable	Status
World coordinate system	\$WORLD	write-protected
Robot coordinate system	\$ROBROOT	write-protected (can be changed in R1/MADA/\$MACHINE.DAT)
Tool coordinate system	\$TOOL*	writable
Base (workpiece) coordinate system	\$BASE*	writable

Table 13 Predefined coordinate systems

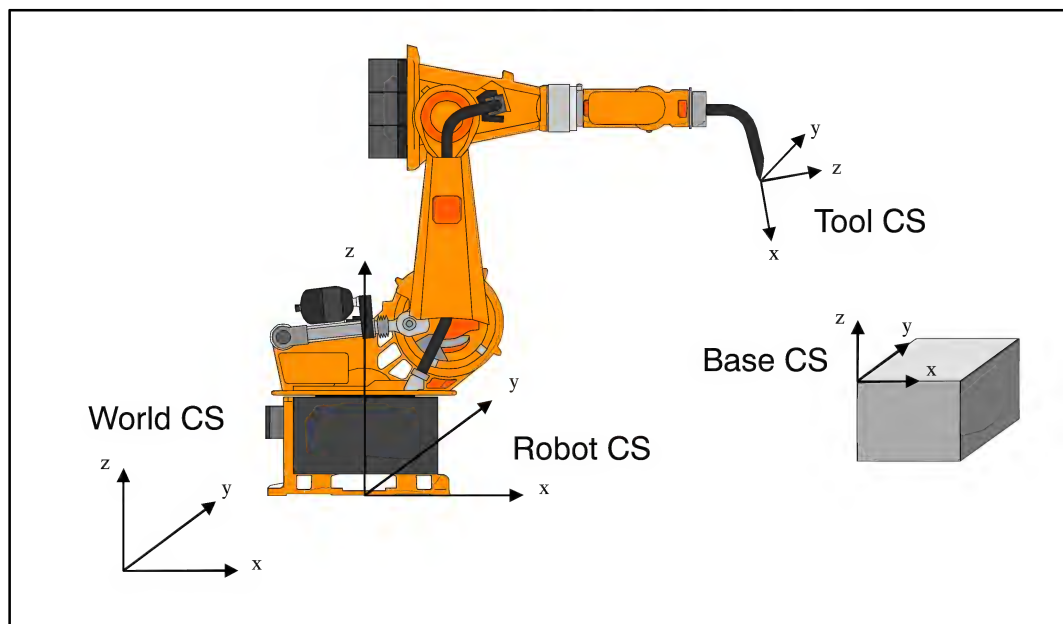


Fig. 12 Cartesian coordinate systems for robots

World coordinate system

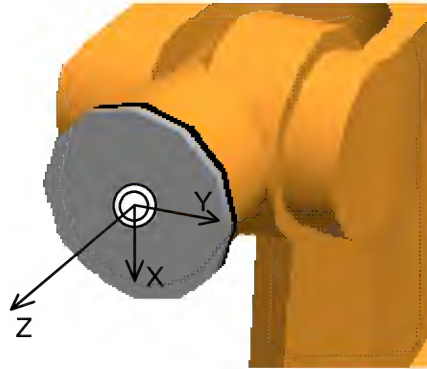
The world coordinate system is a fixed (= does not move when the robot moves) coordinate system, which serves as the underlying coordinate system for a robot system (robot, component support or tool). It represents the reference system for both the robot system and the peripheral equipment of the cell.

Robot coordinate system

This coordinate system is located in the base of the robot and serves as the reference coordinate system for the mechanical construction of the robot. It, in turn, is derived from the world coordinate system and is identical to it when the robot system is delivered. An offset of the robot in relation to \$WORLD can thus be defined using \$ROBROOT.

Tool coordinate system

The tool coordinate system has its origin at the tip of the tool. The orientation can be selected in such a way that its X axis is identical to the tool direction and points out of the tool. If the tool center point is moved, the tool coordinate system is moved with it.



On delivery of the robot, the tool coordinate system is located in the robot flange (the Z axis is identical to axis 6). It is derived, by transformation, from the robot coordinate system.

If a tool change takes place, the original program can still be used after re-calibration since the coordinates of the TCP are known to the computer.

Base coordinate system

The base coordinate system is used as the reference system to define the position of the workpiece. The robot is programmed in the base coordinate system. It has the world coordinate system as its reference coordinate system. At time of delivery, $\$BASE = \$WORLD$. By altering $\$BASE$, it is possible to work, for example, on several identical workpieces in different places using the same program.

Base-related interpolation

When interpolating the motion path, the robot controller calculates, under normal circumstances (tool mounted on the robot flange), the current position ($\$POS_ACT$) in relation to the $\$BASE$ coordinate system (see Fig. 13).

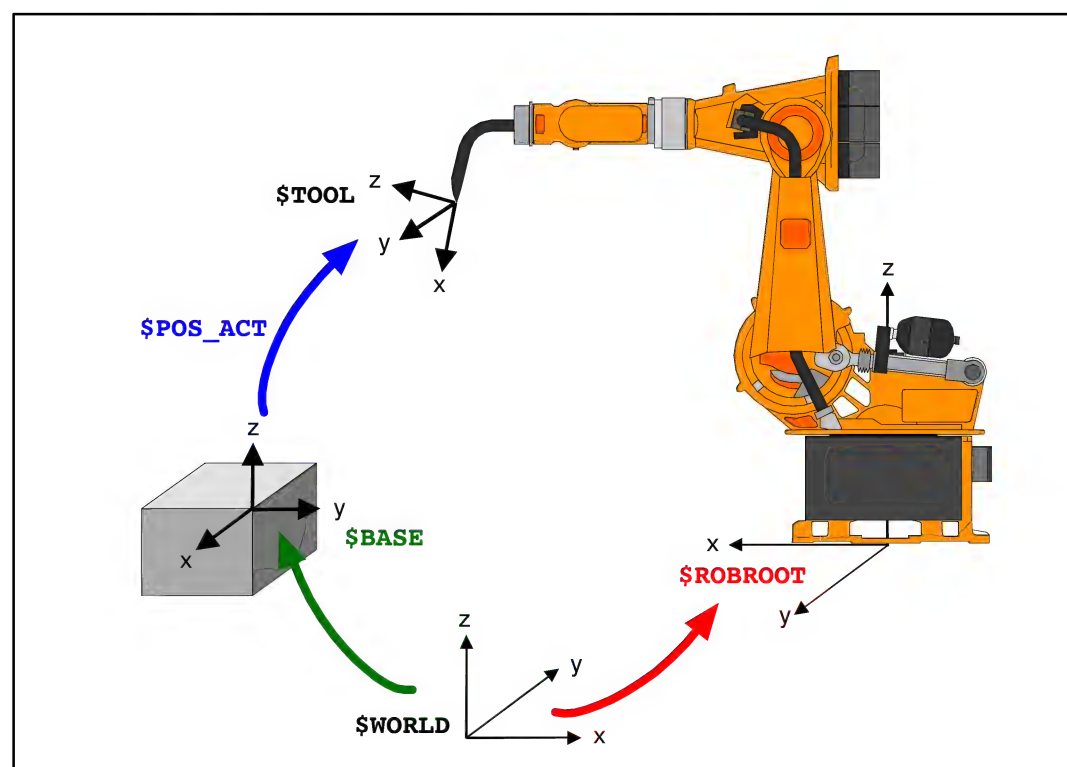


Fig. 13 Kinematic sequence with base-related interpolation

Fixed tool

In industrial practice, however, a gradual switch is being made to anchoring the tool (e.g. welding torch) at a fixed point in space and guiding the workpiece itself, by means of a suitable gripper, along the fixed tool.



The variable **\$TOOL** always refers to the tool or workpiece mounted on the robot. The variable **\$BASE**, on the other hand, always refers to the external tool or workpiece.

Gripper-related interpolation

Since the tool and workpiece have now changed position, but the motion is still to be executed relative to the workpiece, the interpolation of the motion path must now be calculated in the **\$TOOL** coordinate system. This assignment of the interpolation mode occurs implicitly when using a normal or external TCP. This type of interpolation can be defined in the system variable **\$IPO_MODE**. The program line

```
$IPO_MODE = #TCP
```

makes gripper-related interpolation possible in the **\$TOOL** coordinate system. The current position **\$POS_ACT** is thus now calculated relative to **\$TOOL** (see Fig. 14). With

```
$IPO_MODE = #BASE
```

you reset the interpolation mode to base-related interpolation for normal operation. This is also the default setting when the controller is run up.

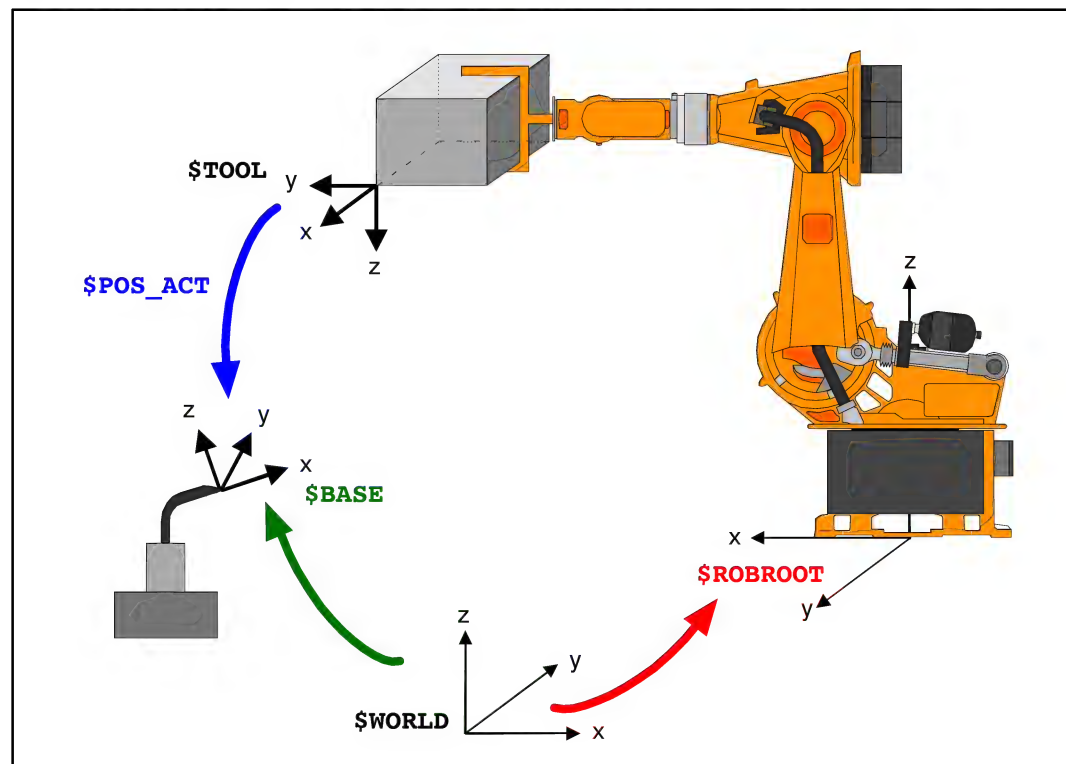


Fig. 14 Kinematic sequence with gripper-related interpolation



An example of the shifting of coordinate systems can be found in the chapter **[Variables and declarations]**, section **[Geometric operator]**.

3.2 Point-to-point motions (PTP)

3.2.1 General (Synchronous PTP)

PTP The point-to-point motion (PTP) is the quickest way of moving the tip of the tool (Tool Center Point: TCP) from the current position to a programmed end position. To do this, the controller calculates the necessary angle differences for each axis.

The following system variables are used

- **\$VEL_AXIS**[*axis number*]: to program maximum axis-specific velocities, and
- **\$ACC_AXIS**[*axis number*]: to program maximum axis-specific acceleration rates.

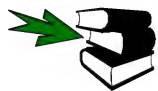
All entries are given as percentages of the maximum value defined in the machine data. If these two system variables have not been programmed for all axes, execution of the program will cause a corresponding error message to be generated.

The movements of the axes are synchronized in such a way (synchronous PTP) that all of the axes start and stop moving at the same time. This means that only the axis with the longest trajectory, the so-called leading axis, is actually moved with the programmed limit value for acceleration and velocity. All other axes move only with the velocity and acceleration rates necessary for them to reach the end point of the motion at the same moment, irrespective of the values programmed in **\$VEL_AXIS**[*No*] and **\$ACC_AXIS**[*No*]. If acceleration adaptation or the higher motion profile is activated (**\$ADAP_ACC**=#STEP1, **\$OPT_MOVE**=#STEP1), axis traversing is additionally phase-synchronous, i.e. all axes enter the acceleration, constant motion and deceleration phases together.



Since it is generally unknown, in the case of PTP motions with Cartesian end coordinates, which is the leading axis, it is usually sensible to set acceleration and velocity values which are identical for all axes.

Synchronous motion control diminishes mechanical stress on the robot since the motor and gear torques are reduced for all axes with shorter trajectories. Phase-synchronous motion control gives (additionally) a motion path which, irrespective of the programmed velocity and acceleration, always follows the same course.



Further information can be found in the chapter **[Motion programming]**, section **[Motions with approximate positioning]**.

3.2.2 Higher motion profile

The higher motion profile is thus used as standard for PTP motions. This model brings about a time-optimized motion from the start point to the end point with **individual PTP instructions** and **PTP approximation instructions**. In other words, it is impossible to move faster with the gears and motors available. The permissible torque range is optimally used for every point along the path, and in particular in the constant velocity phase. The velocity is always adapted in such a way that the torques are not exceeded.

Even with approximation instructions, the only effect of a change to the values for velocity or acceleration is a change to the velocity profile along the path. The geometric curve in space remains unchanged.

The velocity assignments and acceleration limit values (given as percentages) can be set individually for each axis. This limit value relates, however, to the acceleration torque of the axis and not directly to the acceleration itself, i.e. an axis acceleration value of 50% does not necessarily halve the acceleration.

3.2.3 Motion commands

The following program example **PTP_AXIS.SRC** represents the smallest KRL program that can be run.



```

DEF PTP_AXIS()           ;the name of the program is PTP_AXIS

$VEL_AXIS[1]=100         ;definition of the axis velocities
$VEL_AXIS[2]=100
$VEL_AXIS[3]=100
$VEL_AXIS[4]=100
$VEL_AXIS[5]=100
$VEL_AXIS[6]=100

$ACC_AXIS[1]=100         ;definition of the axis accelerations
$ACC_AXIS[2]=100
$ACC_AXIS[3]=100
$ACC_AXIS[4]=100
$ACC_AXIS[5]=100
$ACC_AXIS[6]=100

PTP {AXIS: A1 0,A2 -90,A3 90,A4 0,A5 0,A6 0}

END

```



First of all in this program, the axis velocity and acceleration rates are defined. These assignments must be made before a point-to-point motion can be executed.

The robot then moves each axis into position with the angles specified in the AXIS structure. Example: axis 1 to 0°, axis 2 at -90°, axis 3 at 90°, axis 4 at 0°, axis 5 at 0° and axis 6 at 0°.

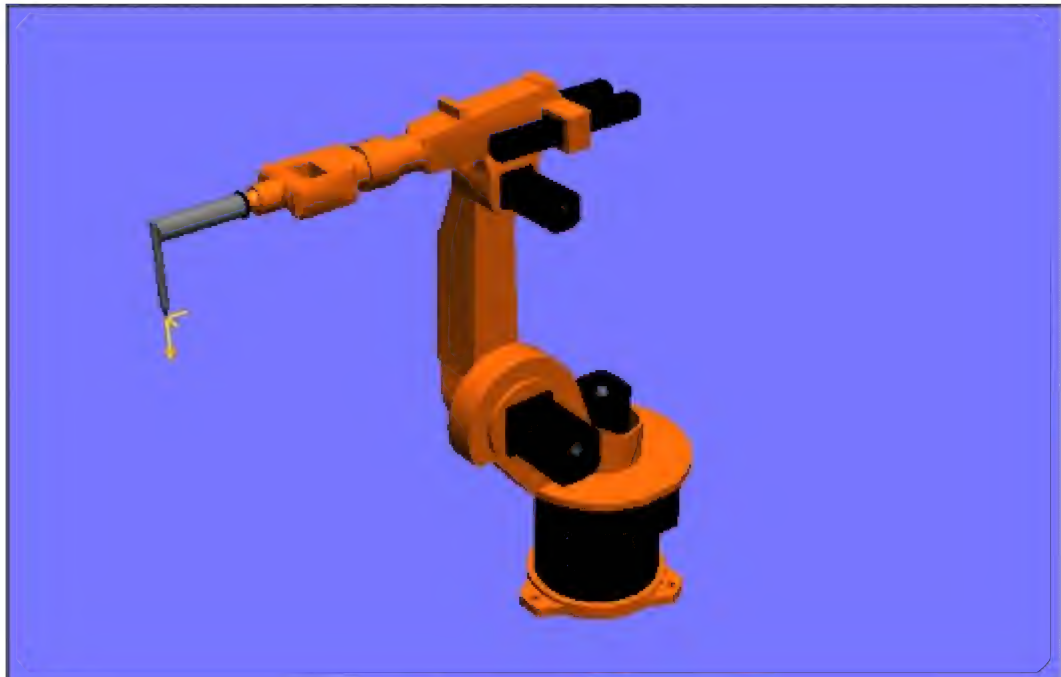


Fig. 15 Mechanical zero position

If individual components are omitted when the axis coordinates are entered, the robot only moves the axes that have been specified; the others do not change position. With

PTP {A3 45}

for example, only axis 3 is moved to 45° . Please note that the angle specifications in the PTP instruction are absolute values. The robot does not, therefore, rotate the axis 45° further, but to the absolute axis position of 45° .

For relative movement, the instruction PTP_REL is used. In order to rotate each of the axes 1 and 4 by 35° , for example, simply program:

PTP_REL {A1 35,A4 35}

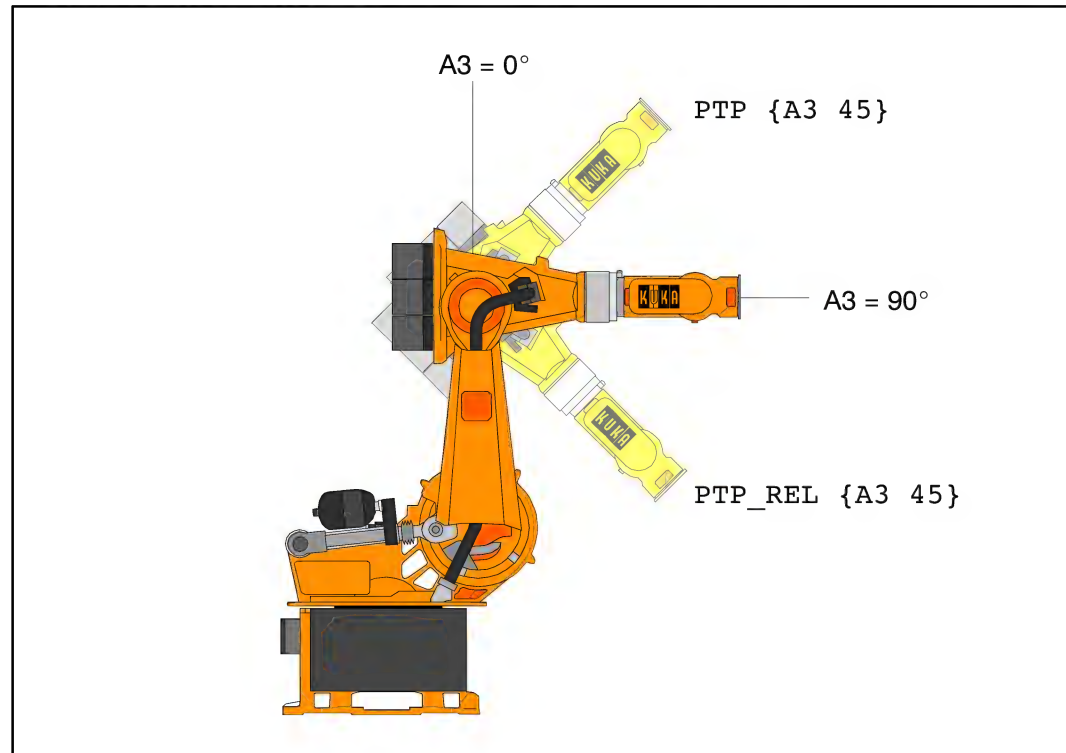


Fig. 16 Difference between absolute and relative axis-specific coordinates



Bear in mind, however, with relative movement, that there is no easy way to resume a motion which has been stopped during execution. After a restart and renewed line selection or change of program run mode, the controller is unable to take the distance already covered into account and moves the robot again by the full relative distance programmed, leading ultimately to an incorrect end point.

Movement using axis-specific coordinates is usually impractical, however, as the human programmer thinks and works in terms of Cartesian space. What is more useful, therefore, is the entry of Cartesian coordinates via a POS structure, as demonstrated in the following example:



```

DEF PTP_POS ( )

$BASE = $WORLD      ;setting of the base coordinate system
$TOOL = $NULLFRAME ;setting of the tool coordinate system

$VEL_AXIS[1]=100    ;definition of the axis velocities
$VEL_AXIS[2]=100
$VEL_AXIS[3]=100
$VEL_AXIS[4]=100
$VEL_AXIS[5]=100
$VEL_AXIS[6]=100

$ACC_AXIS[1]=100    ;definition of the axis accelerations
$ACC_AXIS[2]=100
$ACC_AXIS[3]=100
$ACC_AXIS[4]=100
$ACC_AXIS[5]=100
$ACC_AXIS[6]=100

PTP {POS:X 1025,Y 0,Z 1480,A 0,B 90,C 0,S 'B 010',T 'B 000010'}

END

```

Bear in mind now, when entering end points in Cartesian coordinates, that alongside the entries for velocity and acceleration the base coordinate system and tool coordinate system must also be defined.

Coordinate systems

In our example, the base coordinate system (\$BASE) has been set identical to the world coordinate system (\$WORLD), which is located as standard in the base of the robot (\$ROBROOT). The tool coordinate system (\$TOOL) has been assigned the null frame (\$NULLFRAME = {FRAME: X 0, Y 0, Z 0, A 0, B 0, C 0}), meaning that all entries relate to the flange center point. The tool center point (TCP) is thus, so to speak, also located at the flange center point. If a tool is fitted to the flange, the values must be corrected accordingly. For more information, refer to the documentation on the calibration of tools.

The above PTP instruction now moves the robot in such a way that at the end point of the motion the TCP is shifted 1025 mm in the X direction, 0 mm in the Y direction and 1480 mm in the Z direction from the robot base. Entries A, B and C define the orientation of the TCP. Status S and Turn T define the position of the axes.

Testing this example with a KR 6 robot will produce the same result as in the previous example. The robot moves to the mechanical zero position. Both instructions are thus identical for this model of robot.

It is also possible to omit individual components of the geometrical specification when entering the end point using Cartesian coordinates. The instruction

```
PTP {Z 1300, B 180}
```

moves the TCP in the direction of the Z axis to the absolute position 1300 mm and “tilts” the TCP by 180°.

For relative traversing of the robot the PTP_REL command is used again. With
 PTP_REL {Z 180, B -90}

the robot can then be returned to its original position. Bear in mind again that relative motions, once interrupted, cannot be reselected.

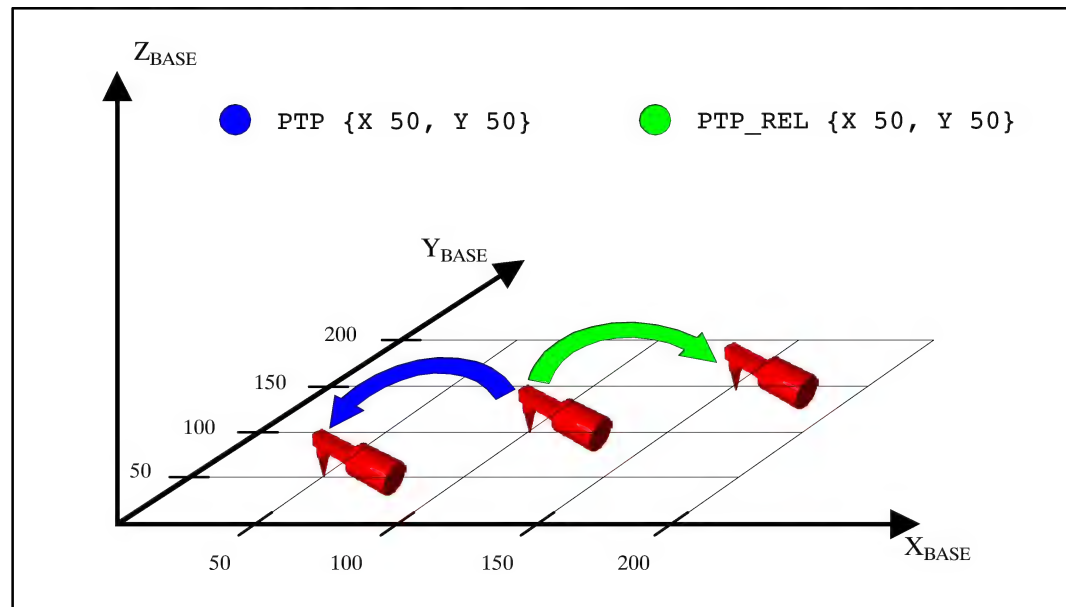


Fig. 17 Difference between absolute and relative Cartesian coordinates



With Cartesian coordinates it is possible to carry out a frame linkage directly in the motion command using the geometric operator. In this way, it is possible, for example, to initiate an offset in relation to the base coordinate system without modifying the variable \$BASE.



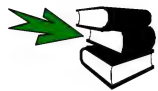
Furthermore, specifying a base offset via the colon operator has a decisive advantage over redefining \$BASE:

The offset occurs in the motion instruction, whereas a \$BASE setting must be made before the motion instruction. In this way, even if the program is stopped and a subsequent block selection made, the correct base for the movement is always selected.

Repeated modification of \$BASE, as shown in the following sequence,

```
...
$BASE = $WORLD
...
PTP POS_1
$BASE = {X 100,Y -200,Z 1000,A 0, B 180,C 45}
PTP POS_2
...
```

on the other hand, would lead to an incorrect end point after cancellation of the POS_2 motion instruction and reselection of the POS_1 instruction, because the POS_1 motion instruction would now also relate to the new base. This also occurs, incidentally, in the event of the first motion instruction being stopped if a corresponding computer advance run is set.



Further information can be found in the chapter **[Motion programming]**, section **[Computer advance run]**.

For this reason, where possible, \$BASE and \$TOOL should only be set once, e.g. in the initialization section of the program. Subsequent offsets can then be carried out using the geometric operator.



When teaching points with the basic package supplied as standard, \$BASE and \$TOOL for each point are automatically stored in the data list.

In the following example, the end point coordinates are shifted 300 mm in the X direction, -100 mm in the Y direction, and rotated by 90° about the Z axis in the second PTP command.



```
DEF FR_VERS ( )

;----- Declaration section -----
EXT BAS (BAS_COMMAND :IN,REAL :IN )
DECL AXIS HOME ;Variable HOME of type AXIS
DECL FRAME BASE1 ;Variable BASE1 of type FRAME

;----- Initialization -----
BAS (#INITMOV,0 ) ;Initialization of velocities,
                  ;accelerations, $BASE, $TOOL, etc.
HOME={AXIS: A1 0,A2 -90,A3 90,A4 0,A5 0,A6 0}
BASE1={FRAME: X 300,Y -100,Z 0,A 90,B 0,C 0}

;----- Main section -----
PTP HOME ; BCO run
; Movement relating to the $BASE coordinate system
PTP {POS: X 540,Y 630,Z 1500,A 0,B 90,C 0,S 2,T 35}
; Movement relating to the $BASE-CS offset by BASE1
PTP BASE1:{POS: X 540,Y 630,Z 1500,A 0,B 90,C 0,S 2,T 35}
PTP HOME
END
```



In this example, moreover, the necessary assignments of velocities and accelerations as well as \$BASE and \$TOOL coordinate systems are no longer carried out "by hand". Instead, the basic package "BAS.SRC", which comes as standard, is used for this. To do this, it must first be made known to the program using the EXT instruction.

The initialization command

INI

```
BAS (#INITMOV, 0)
```

then assigns default values to all the important system variables.



Further information can be found in the chapter **[Subprograms and functions]**, section **[Declaration]**.

BCO

Before a program can be executed, block coincidence (BCO) must be established, i.e. correspondence between the current robot position and the programmed position. Since the BCO run does not represent a programmed, tested motion, it must be executed with the Start key held down ("dead man" function) and at automatically reduced velocity. When the robot has reached the programmed path, the motion is stopped and the program can be continued by pressing the Start key again.



No BCO run is carried out in "Automatic External" mode!

It is thus advisable to program a "Home" run as the first motion instruction; this moves the robot to an unambiguously defined and uncritical initial position in which block coincidence is then also established. The robot should be brought into this position again at the end of the program.

S and T

The entries "S" and "T" in a POS specification serve to select a specific, unambiguously defined robot position where several different axis positions are possible for the same point in space (because of kinematic singularities).

When using Cartesian coordinates it is thus very important also to program "Status" and "Turn" for the first motion instruction in order to define an unambiguous initial position. Since "S" and "T" are not taken into consideration in continuous-path motions (see 3.3), the first motion instruction of a program (home run) must always be a complete PTP instruction with Status and Turn specified (or a complete PTP instruction with axis coordinates).

In subsequent PTP instructions, the entries "S" and "T" can be omitted so long as no specific axis position is required, e.g. because of obstacles. The robot retains the old value for S and selects the T value which gives the shortest possible axis trajectory, which always remains the same, each time the program is run, because of the one-off programming of "S" and "T" in the first PTP instruction.



Status and Turn both require integer entries, which should ideally be made in binary form.

Turn

Expanding a Cartesian position specification to include the "Turn" specification makes it possible to move axes through angles greater than $+180^\circ$ or less than -180° without the need for special movement strategies (e.g. auxiliary points). With rotational axes, the individual bits determine the sign before the axis value in the following way:

Bit x = 0: angle of axis x $\geq 0^\circ$
 Bit x = 1: angle of axis x $< 0^\circ$

Value	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	$A6 \geq 0^\circ$	$A5 \geq 0^\circ$	$A4 \geq 0^\circ$	$A3 \geq 0^\circ$	$A2 \geq 0^\circ$	$A1 \geq 0^\circ$
1	$A6 < 0^\circ$	$A5 < 0^\circ$	$A4 < 0^\circ$	$A3 < 0^\circ$	$A2 < 0^\circ$	$A1 < 0^\circ$

Table 14 Meaning of the Turn bits

The entry T 'B 10011' thus means that the angles of axes 1, 2 and 5 are negative whereas those of axes 3, 4 and 6 are positive (all high-order 0 bits can be omitted).

Status

Status S is used to deal with ambiguities in the axis position (see Fig. 18). S is thus dependent on the current robot kinematic system.

Ambiguities

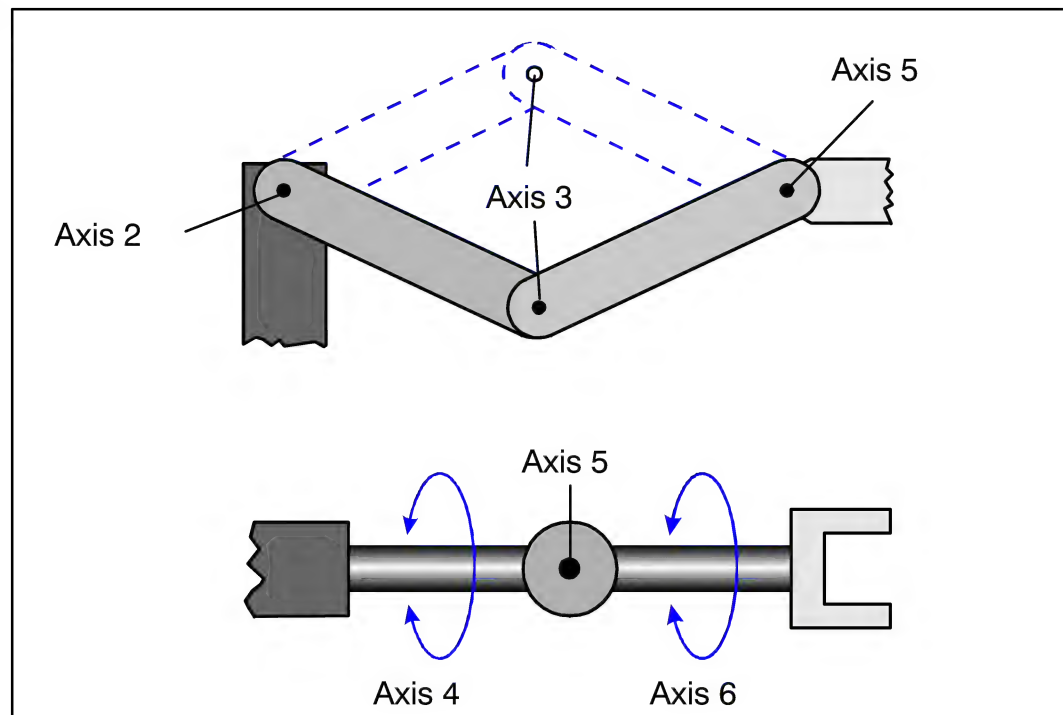


Fig. 18 Examples of ambiguous robot kinematics

The meaning of the individual bits is:

- Bit 0: Position of the wrist root point (basic/overhead area)
- Bit 1: Arm configuration
- Bit 2: Wrist configuration

The bits for all jointed-arm robots with 6 axes are set as shown in the following table:

Value	Bit 2	Bit 1	Bit 0
0	$0^\circ \leq A5 < 180^\circ$ $A5 < -180^\circ$	$A3 < \phi$ (ϕ depends on robot model)	Basic area
1	$-180^\circ \leq A5 < 0^\circ$ $A5 \geq 180^\circ$	$A3 \geq \phi$ (ϕ depends on robot model)	Overhead area

Table 15 Status bits for 6-axis jointed-arm robots

The basic/overhead areas can be visualized in Cartesian terms. To do this, the following terms are defined:

Wrist root point: Intersection of the wrist axes

A1 coordinate system: If axis 1 is at 0° , it is identical to the \$ROBROOT coordinate system. For values not equal to 0° , it moves with axis 1.

The basic/overhead areas can thus be defined as follows:

- If the x-value of the wrist root point, expressed in the A1 coordinate system, is positive, the robot is in the basic area.
- If the x-value of the wrist root point, expressed in the A1 coordinate system, is negative, the robot is in the overhead area.

Bit 1 specifies the position of the arm. The setting of the bit is dependent on the robot model in use. For robots whose axes 3 and 4 intersect, the following applies: bit 1 has the value 0 if $\text{axis } 3 < 0^\circ$, otherwise bit 1 = 1. For robots with an offset between axis 3 and axis 4 (e.g. KR 30, see Fig. 19), the angle at which the value of bit 1 changes depends on the size of this offset.

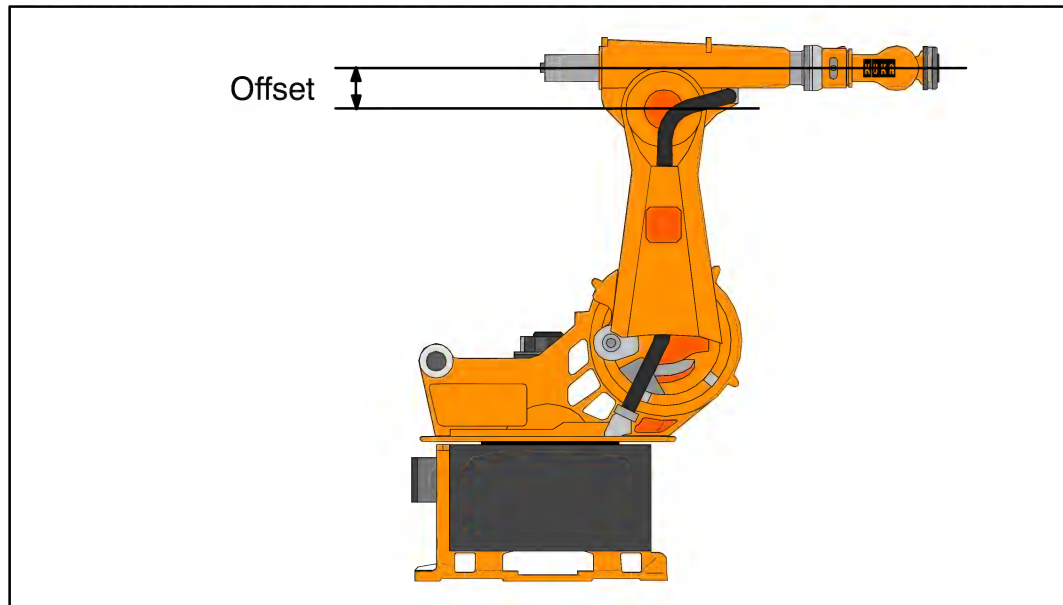


Fig. 19 Offset between axes 3 and 4 of a KR 30

The effects of the status bits on the robot configuration are illustrated in Fig. 20. The robot has been moved to the same point in space with the axes in four different positions. In the first configuration the robot is in a basic position; axis 5 has a value of approx. 45° , axis 3 approx. 80° .

The second robot configuration is barely distinguishable from the first. Axis 4 has simply been rotated by 180° and the other axes realigned accordingly. So, while the configuration of the arm remains the same, that of the wrist has changed: Axis 5 is now at approx. -45° , status bit 2 is therefore 1.

From position 2 to position 3 the arm configuration now changes. Axis 3 rotates to a position with an angle of approx. -50° , status bit 1 takes the value 0.

In the fourth configuration, the robot is finally in the overhead position. To get here, axis 1, in particular, has been rotated by 180° . Status bit 0 becomes 1.

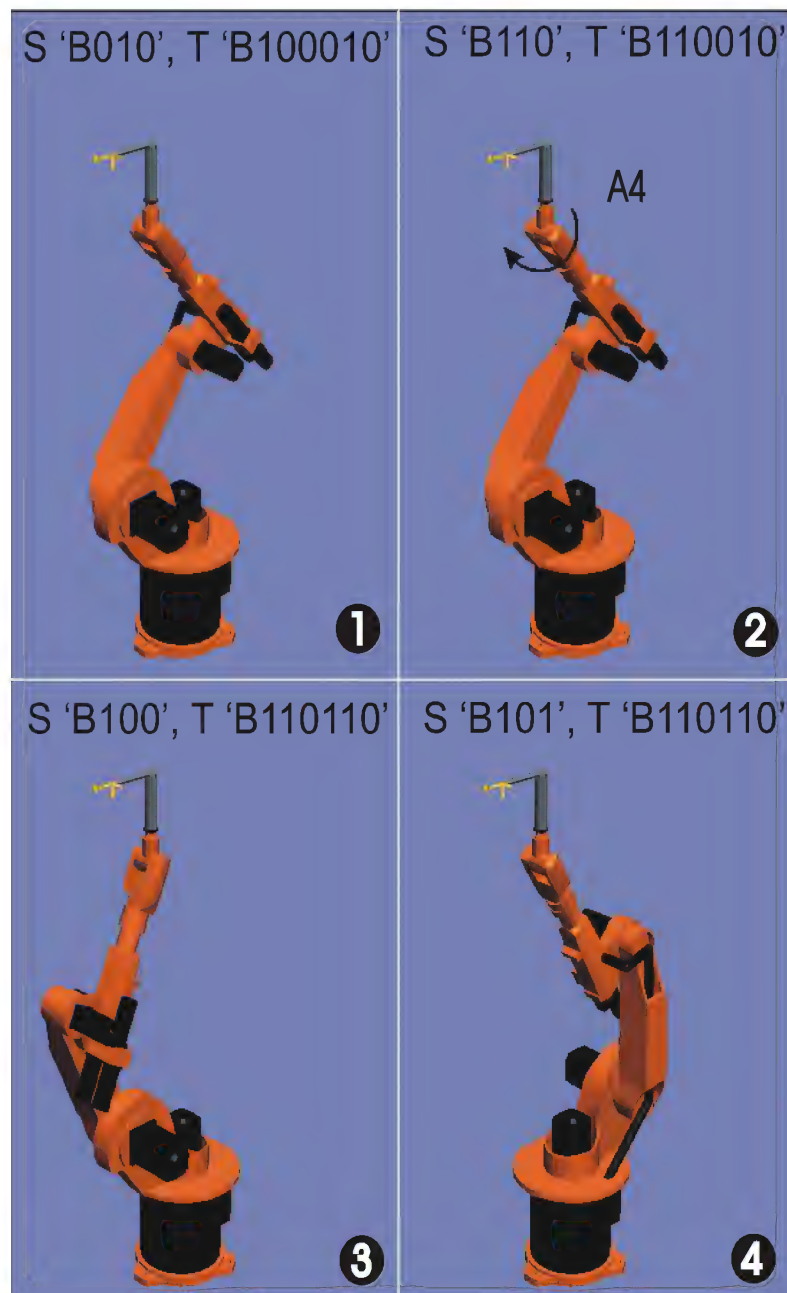


Fig. 20 Effects of the status bits on the position of the robot

3.3 Continuous-path motions (CP motions = Continuous Path)

3.3.1 Velocity and acceleration

Unlike with PTP motions, it is not just start and end positions that are predefined in the case of continuous-path motions. Additionally, movement of the TCP along a linear or circular path between these points is also required.

The velocities and rates of acceleration to be entered do not relate any longer, therefore, to the individual axes, but to the motion of the TCP. The TCP is thereby moved at a precisely defined velocity. Velocities and rates of acceleration must be programmed for the translation, swivel angle and angle of rotation. Table 16 provides an overview of the system variables to be programmed and their units.

	Variable name	Data type	Unit	Function
Velocities	\$VEL.CP	REAL	m/s	Path velocity
	\$VEL.ORI1	REAL	°/s	Swivel velocity
	\$VEL.ORI2	REAL	°/s	Rotational velocity
Accelerations	\$ACC.CP	REAL	m/s ²	Path acceleration
	\$ACC.ORI1	REAL	°/s ²	Swivel acceleration
	\$ACC.ORI2	REAL	°/s ²	Rotational acceleration

Table 16 System variables for CP velocities and accelerations



At any given moment during execution of the motion, at least one of the motion components – translation, swivelling or rotation – is being carried out at a programmed rate of acceleration or velocity. The non-dominant components are phase-synchronously adapted.



When the initialization sequence of the basic package is called, the default settings for the velocities and rates of acceleration of CP motions are also preset to the maximum values defined in the machine data or \$CONFIG.DAT.

The axis velocity and acceleration are still monitored for CP motions and, in the event of the monitoring limit values defined in the system variables \$ACC_ACT_MA and \$VEL_ACT_MA being exceeded, a braking reaction is triggered and an error message is generated. As standard, these limits are set to 250% of the nominal axis acceleration and 110% of the nominal axis velocity. These monitoring ranges are valid for all operating modes and manual traversing.

It is possible, using the system variable \$CP_VEL_TYPE, to reduce the axis feed rates, and thus the acceleration and velocity, in order to prevent a response from the monitoring limits (braking reaction). The default setting for this variable is #CONSTANT, i.e. reduction is not active in program mode. The value #VAR_T1 must be set if this function is required in T1 mode (lower axis velocities and accelerations are used in T1) and the value #VAR_ALL for all other operating modes. Reduction is always active in jog mode.

The system variable \$CPVELREDMELD causes a message to be generated, in both test modes, if the path velocity is reduced. In order to do so, the variable must be assigned the value "1".

3.3.2 Orientation control

If the orientation in space of the tool is to change during the path motion, the orientation control mode can be set using the system variable \$ORI_TYPE (see Fig. 21):

- `$ORI_TYPE = #CONSTANT` During the path motion the orientation remains constant; the programmed orientation is ignored for the end point and that for the start point used.
- `$ORI_TYPE = #VAR` During the path motion the orientation changes continuously from the initial orientation to the end orientation. This value is also set during initialization, by `BAS (#INITMOV, 0)`.

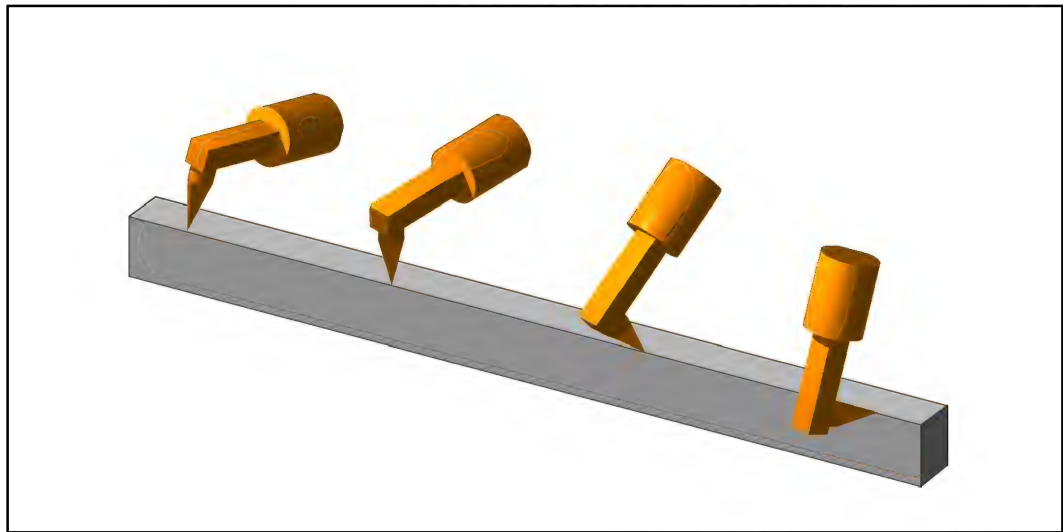


Fig. 21 Change of orientation with a linear motion

With circular motions, in addition to constant and variable orientation, there is a choice between space-related or path-related orientation:

- `$CIRC_TYPE = #BASE` Space-related orientation control during the circular motion. This value is also set during initialization, by `BAS (#INITMOV, 0)`.
- `$CIRC_TYPE = #PATH` Path-related orientation control during the circular motion.

Constant +
path-related

With path-related orientation control, the longitudinal axis of the tool is moved relative to the plane and tangent of the circle. This interrelation can be explained using the so-called tool-based moving frame (see Fig. 22).

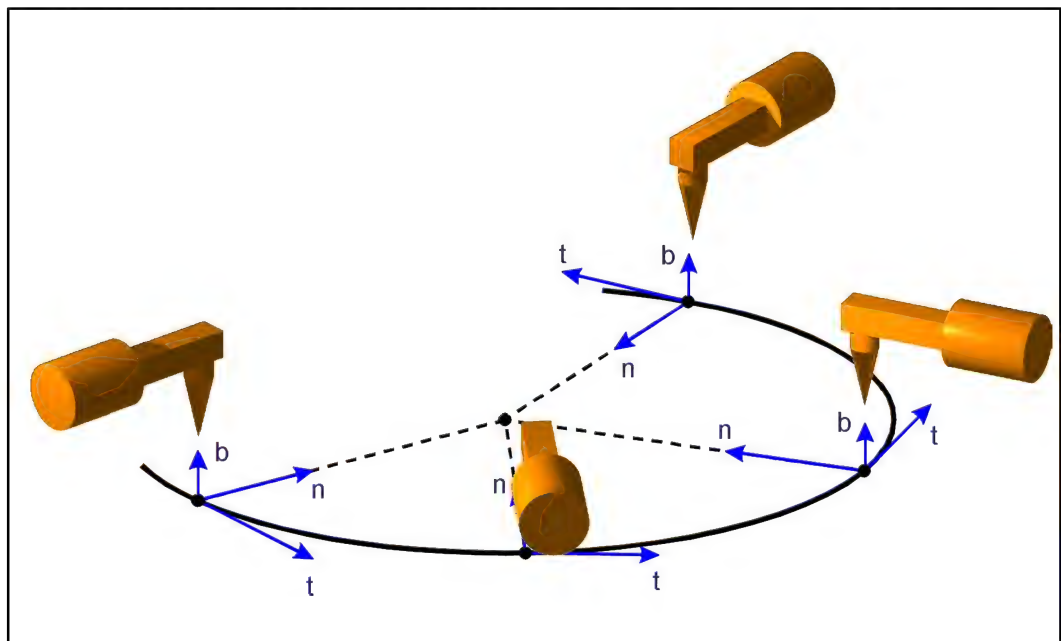


Fig. 22 Constant path-related orientation control with circular motions

The tool-based moving frame comprises the circle tangent vector **t**, the normal vector **n** and the binormal vector **b**. The tool orientation is kept aligned on the circle segment in Fig. 22 by the tool-based moving frame. Relative to the tool-based moving frame, the tool positions undergo no change of orientation. This is an important requirement in arc welding, for example.

In the illustrated example, the tool orientation relative to the tool-based moving frame remains unchanged throughout the motion from the start point to the end point (`$ORI_TYPE=#CONST`).

Variable + path-related If a path-related change in orientation between the start and end points is desired (\$ORI_TYPE=#VAR), this is carried out relative to the tool-based moving frame by means of superposed rotation and swivelling (see Fig. 23). Orientation control in the tool-based moving frame with circular motions is thus perfectly analogous to orientation control with linear motions.

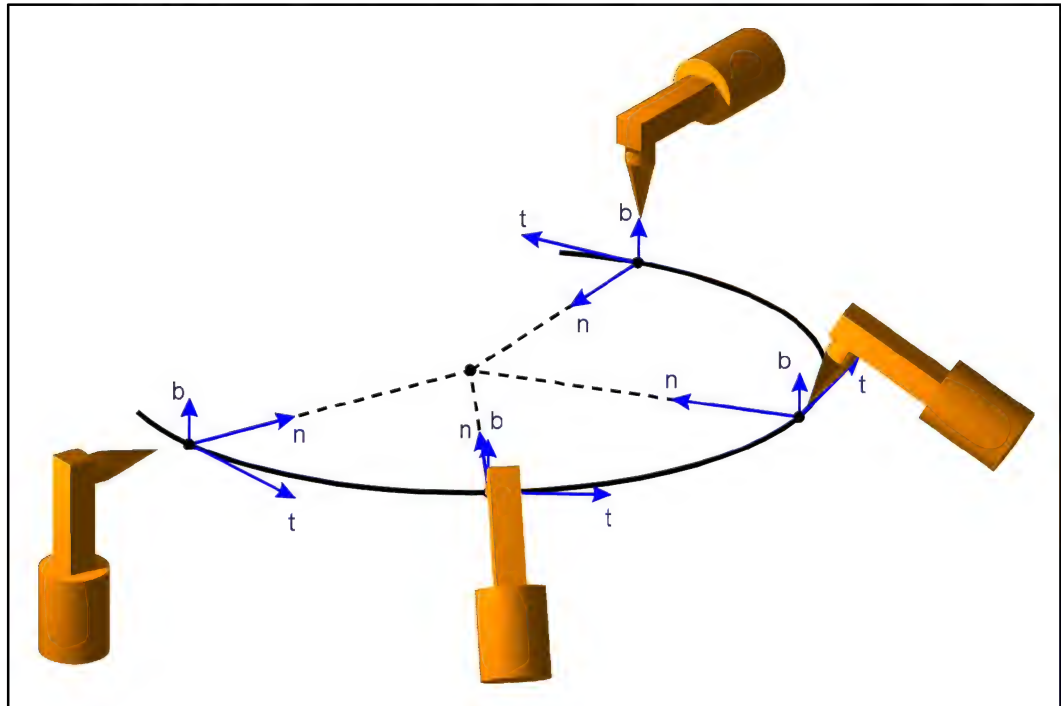


Fig. 23 Variable path-related orientation control with circular motions

Constant +
space-related

With space-related orientation control, the orientation is controlled relative to the current base system (\$BASE).

Space-related orientation control is especially useful for applications where the emphasis is on the path motion, i.e. guiding the TCP along a circular path. This is particularly the case for applications with very little change in orientation between the start and end points or applications where the orientation in space remains exactly constant (see Fig. 24) during a circular motion (e.g. adhesive application with rotationally symmetrical adhesive nozzle).

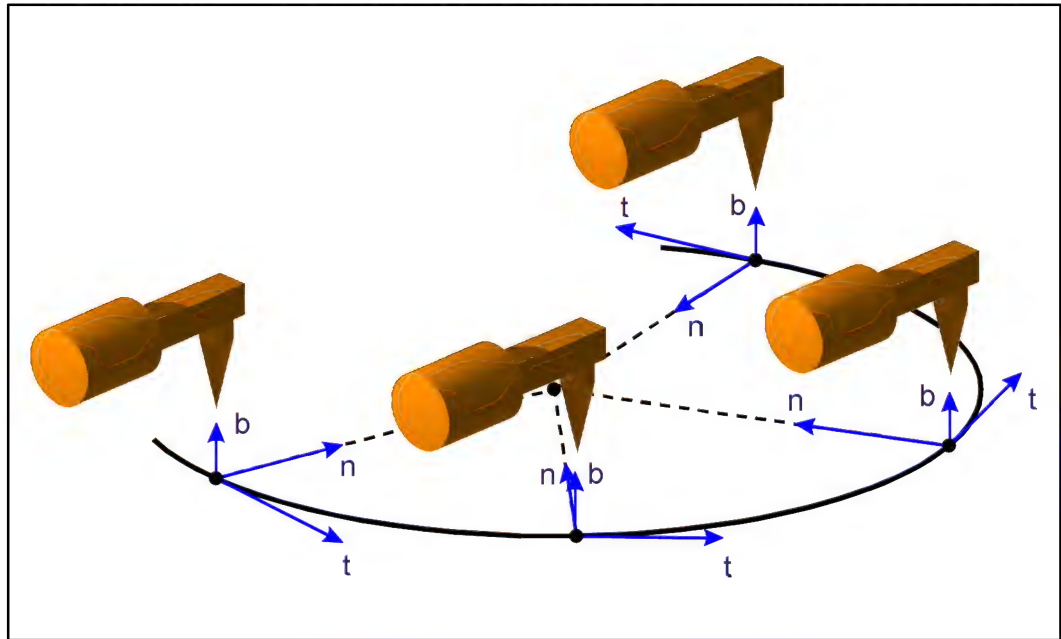


Fig. 24 Constant space-related orientation control with circular motions

Variable +
space-related

A space-related change in orientation ($\$ORI_TYPE=\#VAR$) between the start and end positions is again carried out by the superposition of swivelling and rotational motions (see Fig. 25). In this case, however, it is relative to the base coordinate system.

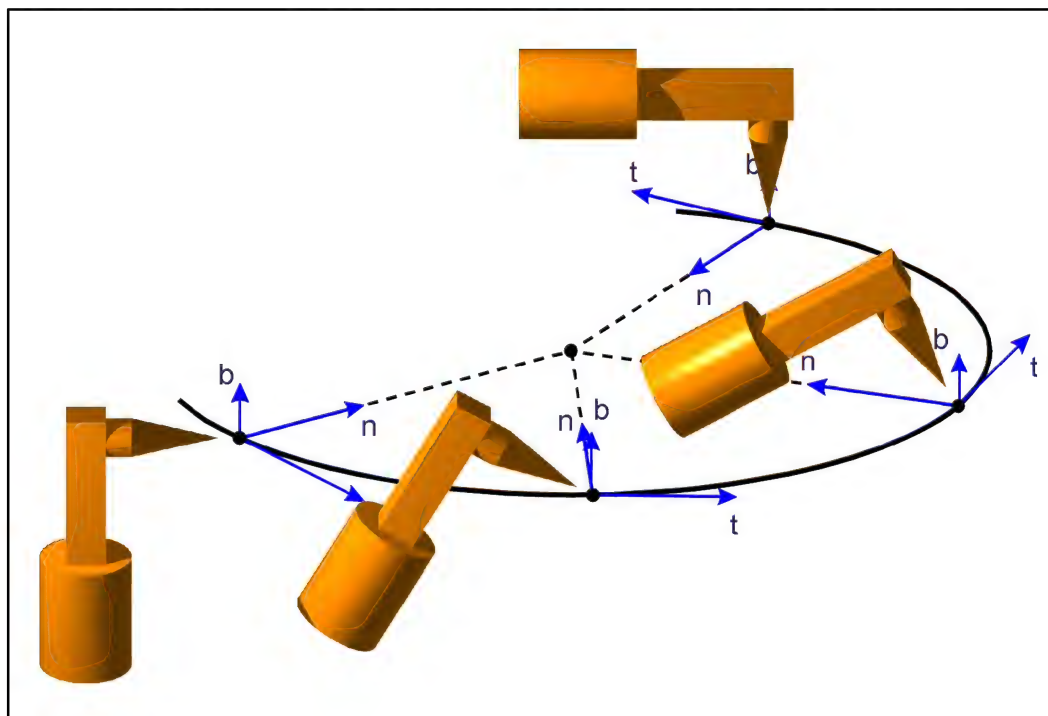


Fig. 25 Variable space-related orientation control with circular motions

The default settings for the system variables used for orientation control with path motions are listed again in Table 17:

	in the system	by BAS ($\#INITMOV, 0$)
$\\$ORI_TYPE$	$\#VAR$	
$\\$CIRC_TYPE$	$\#PATH$	$\#BASE$

Table 17 Default settings of $\$ORI_TYPE$ and $\$CIRC_TYPE$

3.3.3 Linear motions

LIN

In the case of a linear motion, the KR C... calculates a straight line from the current position (the last point programmed in the program) to the position specified in the motion command.

A linear motion is programmed using the keywords `LIN` or `LIN_REL` in connection with the specification of the end point, i.e. analogous to PTP programming. The end position for linear motions is entered with Cartesian coordinates. Only the data types `FRAME` or `POS` are thus permissible.

In the case of linear motions, the angle status of the end point must always be the same as that of the start point. Specification of Status and Turn for an end point of the data type `POS` will thus be ignored. A PTP motion with complete coordinate specification (e.g. HOME run) must therefore be programmed before the first `LIN` instruction.

The assignment of velocity and acceleration variables necessary for continuous-path motions, as well as the setting of tool and base coordinate systems, is again carried out, in the following sample program, using the initialization routine `BAS.SRC`.



```

DEF LIN_BEW ()

;----- Declaration section -----
EXT BAS (BAS_COMMAND: IN, REAL: IN)
DECL AXIS HOME      ;Variable HOME of type AXIS

;----- Initialization -----
BAS (#INITMOV, 0)    ;Initialization of velocities,
                    ;accelerations, $BASE, $TOOL, etc.
HOME = {AXIS: A1 0,A2 -90,A3 90,A4 0,A5 0,A6 0}

;----- Main section -----
PTP HOME ; BCO run
PTP {A5 30}

; Linear motion to the specified position, the orientation
; is continuously changed to the end orientation
LIN {X 1030,Y 350,Z 1300,A 160,B 45,C 130}

; Linear motion in the Y-Z plane, S and T are ignored
LIN {POS: Y 0,Z 800,A 0,S 2,T 35}

; Linear motion to the specified position, the orientation
; is not changed
$ORI_TYPE=#CONST
LIN {FRAME: X 700,Y -300,Z 1000,A 23,B 230,C -90}

; The orientation is still not changed
LIN {FRAME: Z 1200,A 90,B 0,C 0}

; Relative motion along the X axis
LIN_REL {FRAME: X 300}

PTP HOME
END

```


3.3.4 Circular motions

CIRC

To define a circle or arc in space unambiguously, three points are needed which are different from one another and do not lie on a straight line.

The start point of a circular motion is again formed, as with PTP oder LIN, by the current position.

CA

In order to program a circular motion with the instructions CIRC or CIRC_REL, therefore, an auxiliary point must be defined in addition to the end point. When the controller calculates the motion path, only the translational components (X, Y, Z) of the auxiliary point are evaluated. Depending on the orientation control mode, the orientation of the TCP either changes continuously from the start point to the end point or remains constant.



In addition to the auxiliary and end positions it is also possible to program a circular angle using the option CA (Circular Angle). The geometry of the arc is defined, as always, by means of start, auxiliary and end points. The actual end position on the arc, however, where the motion ends, is determined by the programmed circular angle. This is particularly useful for reprogramming the end position without changing the geometry of the circle.

The arc to be covered can be lengthened or shortened according to the circular angle. The programmed end orientation is then reached at the actual end point. The rotation direction, i.e. the direction in which the TCP should move round the arc, can be defined by the sign before the circular angle (see Fig. 26):

- $CA > 0^\circ$ in the programmed direction (start point \rightarrow auxiliary point \rightarrow end point)
- $CA < 0^\circ$ against the programmed direction (start point \rightarrow end point \rightarrow auxiliary point)



The value for the circular angle is unlimited. In particular, full circles ($> 360^\circ$) can be programmed.

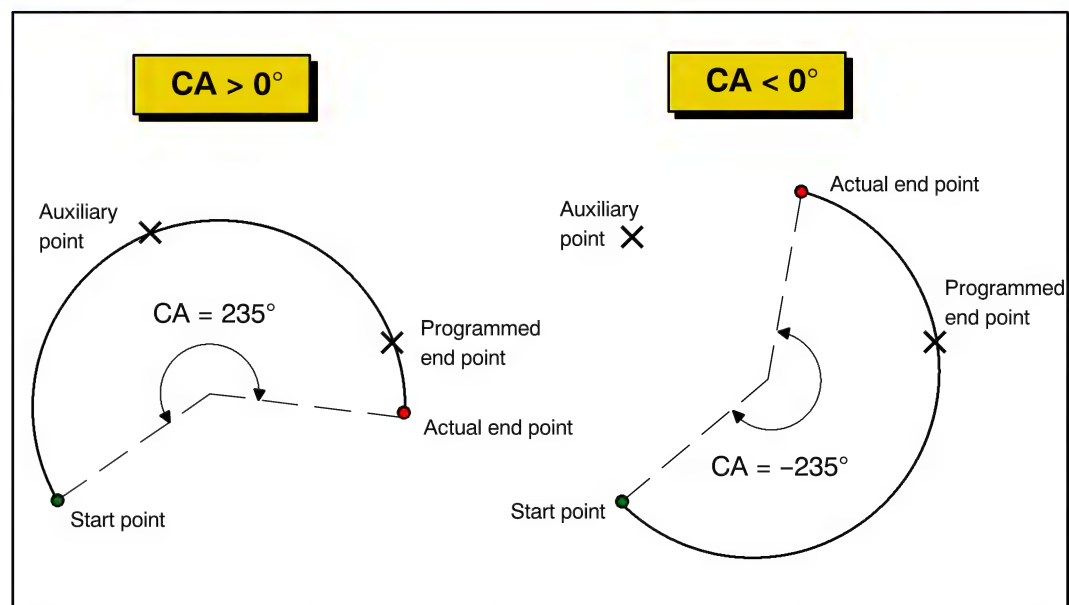


Fig. 26 Effect of the CA option in the CIRC or CIRC_REL command

Relative specifications for auxiliary and end positions (CIRC_REL) relate to the respective start position. As in the case of LIN motions, axis-specific position specifications are not permissible. In the same way, \$BASE and \$TOOL must be fully assigned before executing a circular motion.



```

DEF CIRC_BEW ( )

;----- Declaration section -----
EXT BAS (BAS_COMMAND :IN,REAL :IN )
DECL AXIS HOME

;----- Initialization -----
BAS (#INITMOV,0 ) ;Initialization of velocities,
                  ;accelerations, $BASE, $TOOL, etc.
HOME={AXIS: A1 0,A2 -90,A3 90,A4 0,A5 0,A6 0}

;----- Main section -----
PTP HOME ;BCO run
PTP {POS: X 980,Y -238,Z 718,A 133,B 66,C 146,S 6,T 50}

; Space-related variable orientation control (default setting)
CIRC {X 925,Y -285,Z 718},{X 867,Y -192,Z 718,A 155,B 75,C 160}

; Space-related constant orientation control
; End point defined by angle specification
$ORI_TYPE=#CONST
CIRC {X 982,Y -221,Z 718,A 50,B 60,C 0},{X 1061,Y -118,Z 718,
A -162,B 60,C 177}, CA 300.0

; Path-related constant orientation control
; End point defined by angle specification (backwards)
$CIRC_TYPE=#PATH
CIRC {X 867,Y -192,Z 718},{X 982,Y -221,Z 718,A 0}, CA -150

$ORI_TYPE=#VAR
LIN {A 100} ; Reorientation of the TCP

; Path-related variable orientation control
CIRC {X 963.08,Y -85.39,Z 718},{X 892.05,Y 67.25,Z 718.01,
A 97.34,B 57.07,C 151.11}

; Relative circular motion
CIRC_REL {X -50,Y 50},{X 0,Y 100}

PTP HOME
END

```

3.4 Computer advance run

A highly important performance feature of an industrial robot is the rapidity with which it can complete its work. The efficacy with which the robot can process the application program, comprising not only motions, but also arithmetic and peripheral-controlling instructions, is, along with the robot dynamics, of major significance in this respect.

Faster processing of programs can be achieved

- by reducing the duration of robot motions and
- by shortening the idle time between motions.

Where constraints exist, such as maximum axis velocities and acceleration rates, the first of these can be achieved by the time-optimized approximation of movements.



Further information can be found in the chapter **[Motion programming]**, section **[Motions with approximate positioning]**.

The idle time between motions can be shortened by executing the time-consuming arithmetic and logic instructions between motion commands while the robot is moving, i.e. processing them during the advance run (the instructions “run” in “advance” of the motion).

\$ADVANCE

Using the system variable \$ADVANCE it is possible to define the maximum number of motion blocks the advance run may process ahead of the main run (the motion block currently being executed). The main run pointer, which can be seen on the graphic user interface when the program is running, always indicates the motion command currently being processed.

The advance run pointer, on the other hand, is not visible and can indicate both instructions which are executed completely by the controller and motion blocks which are only prepared by the controller and executed later in the main run (see Fig. 27).

```

13
14  $ADVANCE=1
15
16  →LIN {X 1620,Y 0,Z 1910,A 0,B 90,C 0} ← Main run pointer
17
18  STROM={STROM*1.2}/0.5
19  FOR I=1 TO 6
20      $VEL_AXIS[I]=60
21      $ACC_AXIS[I]=35
22  ENDFOR
23
24  PTP PUNKT6 ← Advance run pointer is located here;
25              $ADVANCE = 1
26  SPANNUNG=110
27
28  PTP PUNKT7
29
30
31

```

Fig. 27 Main run and advance run pointers

In the program extract shown above, the advance run is set to 1 and the main run pointer is positioned in line 16 (i.e. the LIN motion is currently being executed). A computer advance run of 1 means that the instructions in lines 16 to 22 have been completely processed parallel to the execution of the motion and that the motion data for the PTP movement in line 24 are currently being prepared.



To make an approximation possible, a computer advance run of at least 1 must be set. (The variable \$ADVANCE has the value "3" by default. A maximum of 5 advance run steps is possible.)

No computer advance run is possible in an interrupt subprogram. The controller always processes interrupt programs line by line; for this reason approximation is not possible in interrupt programs.

Default settings of \$ADVANCE:

	in the system	by BAS (#INITMOV, 0)
\$ADVANCE	0	3

**Automatic
advance
stop** **run**

Instructions and data which affect the peripheral equipment (e.g. I/O instructions), or which are based on the current state of the robot, trigger an **advance run stop** (see Table 18). This is necessary in order to guarantee the correct sequence of instructions and robot motions.

Instructions	ANOUT ON	ANOUT OFF		
	ANIN ON	ANIN OFF		
	DIGIN ON	DIGIN OFF		
	PULSE			
	HALT	WAIT		
	CREAD	CWRITE	COPEN	CCLOSE
	SREAD	SWRITE		
	CP-PTP combination without approximate positioning			
Instructions combined with an interrupt	END (if a non-global interrupt has been defined in the module)			
	INTERRUPT DECL (if the interrupt has already been declared)			
	RESUME without BRAKE			
Customary system variables	\$ANOUT[Nr]	\$ANIN[Nr]		
	\$DIGIN1	\$DIGIN2	...	\$DIGIN6
	\$OUT[Nr]	\$IN[Nr]		
	\$AXIS_ACT	\$AXIS_BACK	\$AXIS_FOR	\$AXIS_RET
	\$POS_ACT	\$POS_BACK	\$POS_FOR	\$POS_RET
	\$AXIS_INC	\$AXIS_INT	\$POS_ACT_MES	\$POS_INT
	\$TORQUE_AXIS	\$ASYNC_AXIS		
	\$TECH[X].MODE, \$TECH[X].CLASS for certain operations			
	\$LOAD, \$LOAD_A1, \$LOAD_A2, \$LOAD_A3 (in the case of an absolutely accurate robot with a change of load)			

Further system variables	\$ALARM_STOP \$AXIS_ACTMOD \$INHOME_POS \$ON_PATH \$EM_STOP \$EXTSTARTTYP \$REVO_NUM \$SAFETY_SW \$ACT_TOOL \$PAL_MODE \$ACT_BASE \$ACT_EX_AX \$OV_PRO \$WORKSPACE \$IBUS_OFF \$IBUS_ON \$ASYNC_EX_AX _DECOUPLE
Imported variables	All, when accessed
Miscellaneous	In the event of a change of filter between approximated blocks, an advance run stop is triggered.

Table 18 Instructions and variables which trigger an automatic advance run stop

CONTINUE

In applications where this advance run stop should be prevented, the command `CONTINUE` must be programmed immediately before the relevant instruction. The controller then allows the advance run to continue. The effect of this command is limited to the next program line (even if this line is empty!!).



If, on the other hand, you want to stop the advance run at a specific point, without having to alter the system variable `$ADVANCE`, you can make use of a little trick: simply program a wait time of 0 seconds at this point. The instruction `WAIT` automatically stops the advance run, but does nothing else:

```
WAIT SEC 0
```



Further information can be found in the chapter **[Program execution control]**, section **[Wait times]**.

3.5 Motions with approximate positioning

In order to increase velocity, points for which exact positioning is not necessary can be approximated. The robot takes a shortcut as illustrated in Fig. 28.

Approximate
positioning
contour

The approximate positioning contour is automatically generated for this by the controller. The programmer can only influence the beginning and the end of approximate positioning. To calculate the approximation instruction, the controller needs the data for the start point, the approximate positioning point and the end point.

To make approximate positioning possible, a computer advance run (\$ADVANCE) of at least 1 must be set. If the computer advance run is too small, the message "Approximation not possible" appears and the robot is positioned exactly at each point.

If, after an approximate positioning instruction, you program an instruction which automatically stops the advance run (see Table 18), approximate positioning is not possible.



Further information on the TRIGGER instruction as a remedy can be found in the chapter **[Trigger – Path-related switching actions]**.

Approximate
positioning

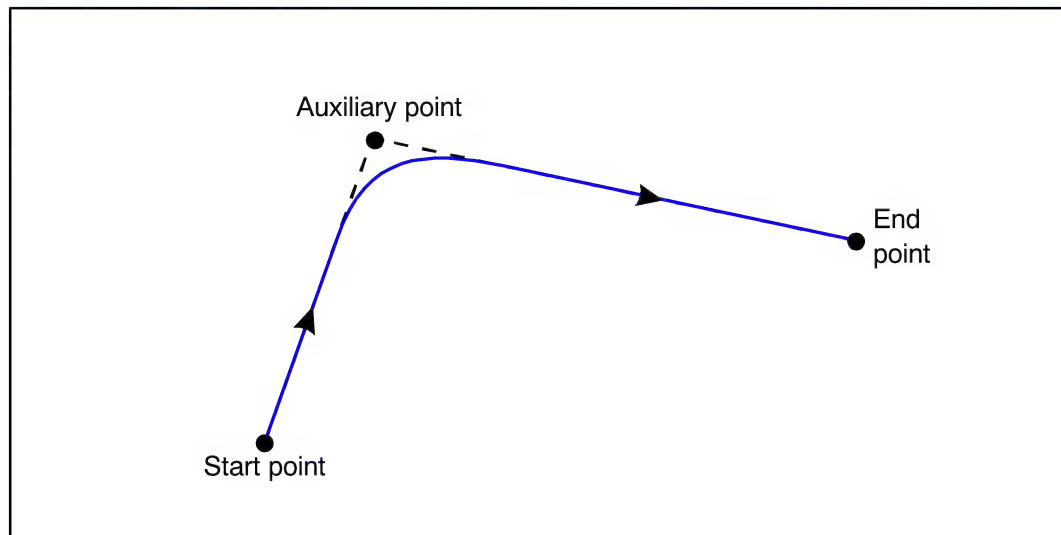


Fig. 28 Approximate positioning at auxiliary points

3.5.1 PTP-PTP approximate positioning

For the purposes of PTP approximate positioning, the controller calculates the distances the axes are to move in the approximate positioning range and plans velocity profiles for each axis which ensure tangential transition from the individual instructions to the approximate positioning contour.

Start of
approximate
positioning

Approximate positioning begins when the last (= leading) axis falls below a specified angle to the approximate positioning point. An angle is predefined for each axis in the machine data.

```
$APO_DIS_PTP[1] = 90
```

```
⋮
```

```
$APO_DIS_PTP[6] = 90
```

In the program, `$APO.CPTP` enables the start of approximate positioning to be specified as a percentage of these maximum values. For example:

```
$APO.CPTP = 50
```

In this example, approximate positioning is begun when the first axis has a residual angle of 45° (50% of 90°) to cover to the approximate positioning point. Approximate positioning ends at the exact moment the first axis has covered an angle of 45° from the approximate positioning point.

The bigger `$APO.CPTP` is, the more the path is rounded.

Approximate positioning can never take place over the middle of the block! In such a case, the system independently limits itself to the middle of the block.

C_PTP

The approximate positioning of a point is displayed in the PTP command by adding the keyword `C_PTP`:

```
PTP POINT4 C_PTP
```

The PTP approximate positioning instruction too is executed in a time-optimized manner, i.e. during approximate positioning, there is always at least one axis moving with the programmed acceleration or velocity limits. The system simultaneously ensures that the permissible gear and motor torques for each axis are not exceeded. Furthermore, the higher motion profile, set by default, ensures motion execution free from velocity and acceleration variation.



Further information can be found in the chapter **[Motion programming]**, section **[Higher motion profile]**.

From the following example, you can see the effects of the approximate positioning instruction and the variable `$APO.CPTP`. The path covered is illustrated in the x-y plane in Fig. 29. Particularly apparent in this diagram is the fact that the TCP does not move in a straight line between the end points in PTP motions.



```

DEF  UEBERPTP ( )

;----- Declaration section -----
EXT  BAS (BAS_COMMAND :IN,REAL :IN )
DECL AXIS HOME

;----- Initialization -----
BAS (#INITMOV,0 ) ;Initialization of velocities,
                  ;accelerations, $BASE, $TOOL, etc.
HOME={AXIS: A1 0,A2 -90,A3 90,A4 0,A5 0,A6 0}

;----- Main section -----
PTP  HOME ; BCO run

PTP {POS:X 1159.08,Y -232.06,Z 716.38,A 171.85,B 67.32,C
162.65,S 2,T 10}

;Approximate positioning of the point
PTP {POS:X 1246.93,Y -98.86,Z 715,A 125.1,B 56.75,C 111.66,S 2,T
10} C_PTP

PTP {POS:X 1109.41,Y -0.51,Z 715,A 95.44,B 73.45,C 70.95,S 2,T
10}

;Approximate positioning of two points
$APO.CPTP=20
PTP {POS:X 1296.61,Y 133.41,Z 715,A 150.32,B 55.07,C 130.23,S
2,T 11} C_PTP
PTP {POS:X 988.45,Y 238.53,Z 715,A 114.65,B 50.46,C 84.62,S 2,T
11} C_PTP

PTP {POS:X 1209.5,Y 381.09,Z 715,A -141.91,B 82.41,C -159.41,S
2,T 11}

PTP  HOME
END

```

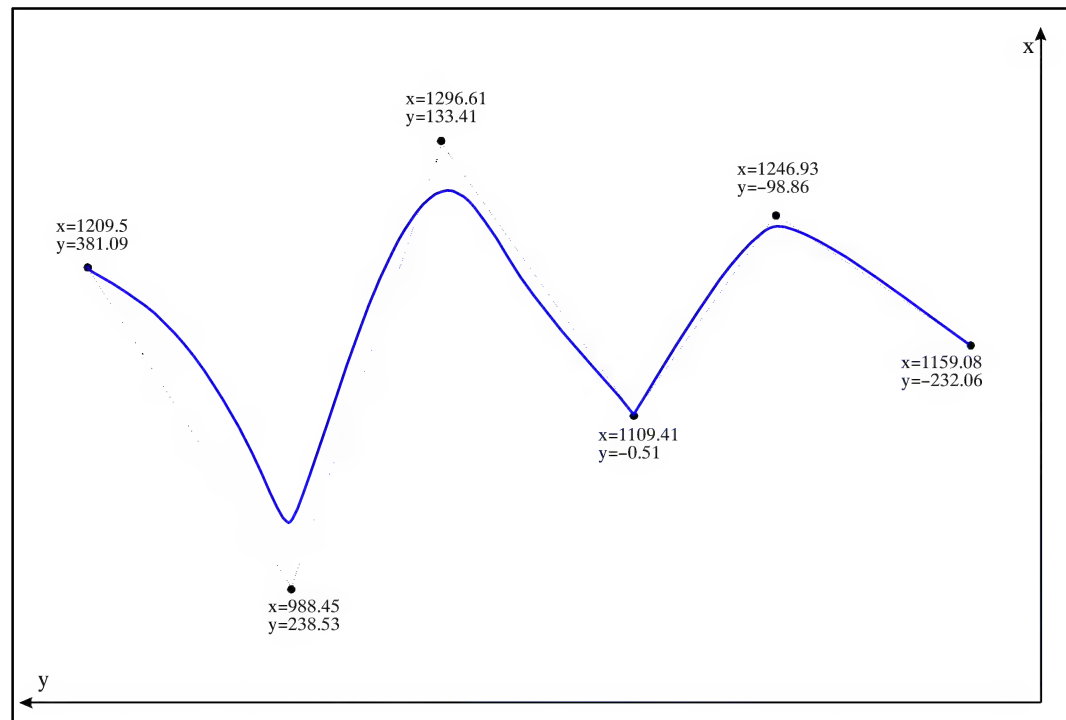


Fig. 29 Example of PTP-PTP approximate positioning



Since the path of a PTP motion is generally neither a straight line nor situated in a plane in space, it should not actually, strictly speaking, be represented as in Fig. 29. Despite the fact that the z value for all points in the example is identical, not every point on the motion path lies in the plane $z=715$ mm. The illustrated path is thus only a projection, in the x-y plane, of the actual path.

3.5.2 LIN–LIN approximate positioning

In order to achieve continuous motion along complex paths, approximate positioning between individual linear blocks is also necessary. The various orientation motions in the LIN blocks must succeed one another smoothly here also



The controller calculates a parabolic path for the approximate positioning contour as this contour form provides a very close approximation of the path of the individual blocks with optimal use of acceleration reserves in the approximate positioning range.

Start of
approximate
positioning

Three predefined variables are available for specifying the start of approximate positioning (see Table 19):

Variable	Data type	Unit	Meaning	Keyword in the command
\$APO.CDIS	REAL	mm	Translational distance criterion	C_DIS
\$APO.CORI	REAL	°	Orientation distance	C_ORI
\$APO.CVEL	INT	%	Velocity criterion	C_VEL

Table 19 System variables for defining the start of approximate positioning

Distance
criterion

A translational distance can be assigned to the variable **\$APO.CDIS**. If the approximate positioning is triggered by this variable, the controller leaves the individual block contour at the exact moment the distance from the end point falls below the value in **\$APO.CDIS**.

Orientation
criterion

An orientation distance can be assigned to the variable **\$APO.CORI**. In this case, the individual block contour is left at the exact moment the dominant orientation angle (swivelling or rotation of the longitudinal tool axis) falls below the angle distance, defined in **\$APO.CORI**, from the programmed approximate positioning point.

Velocity
criterion

A percentage value can be assigned to the variable **\$APO.CVEL**. This value specifies the percentage of the programmed velocity (**\$VEL**) at which the approximate positioning process is started in the deceleration phase of the individual block. The component which, during the motion, reaches or comes closest to the programmed velocity value, is then evaluated in terms of translation, swivel and rotation.

The larger the values in **\$APO.CDIS**, **\$APO.CORI** or **\$APO.CVEL**, the earlier the approximate positioning begins.

In certain circumstances, the system may shorten approximate positioning (middle of the block, symmetry criterion), but will never lengthen it.

C_DIS
C_ORI
C_VEL

Approximate positioning is activated by inserting one of the keywords **C_DIS**, **C_ORI** or **C_VEL** into the LIN or LIN_REL instruction.

The following example serves to illustrate this in conjunction with Fig. 30:



```

DEF  UEBERLIN ( )

;----- Declaration section -----
EXT  BAS (BAS_COMMAND :IN,REAL :IN )
DECL AXIS HOME

;----- Initialization -----
BAS (#INITMOV,0 ) ;Initialization of velocities,
                  ;accelerations, $BASE, $TOOL, etc.
HOME={AXIS: A1 0,A2 -90,A3 90,A4 0,A5 0,A6 0}

;----- Main section -----
PTP  HOME ; BCO run

PTP  {POS: X 1159.08,Y -232.06,Z 716.38,A 171.85,B 67.32,C
162.65,S 2,T 10}

;Approximate positioning of the point using distance criterion
$APO.CDIS=20
LIN  {X 1246.93,Y -98.86,Z 715,A 125.1,B 56.75,C 111.66} C_DIS
LIN  {X 1109.41,Y -0.51,Z 715,A 95.44,B 73.45,C 70.95}

;Approximate positioning of two points
LIN  {X 1296.61,Y 133.41,Z 714.99,A 150.32,B 55.07,C 130.23}
C_ORI
LIN  {X 988.45,Y 238.53,Z 714.99,A 114.65,B 50.46,C 84.62} C_VEL

LIN  {X 1209.5,Y 381.09,Z 715,A -141.91,B 82.41,C -159.41}

PTP  HOME
END

```

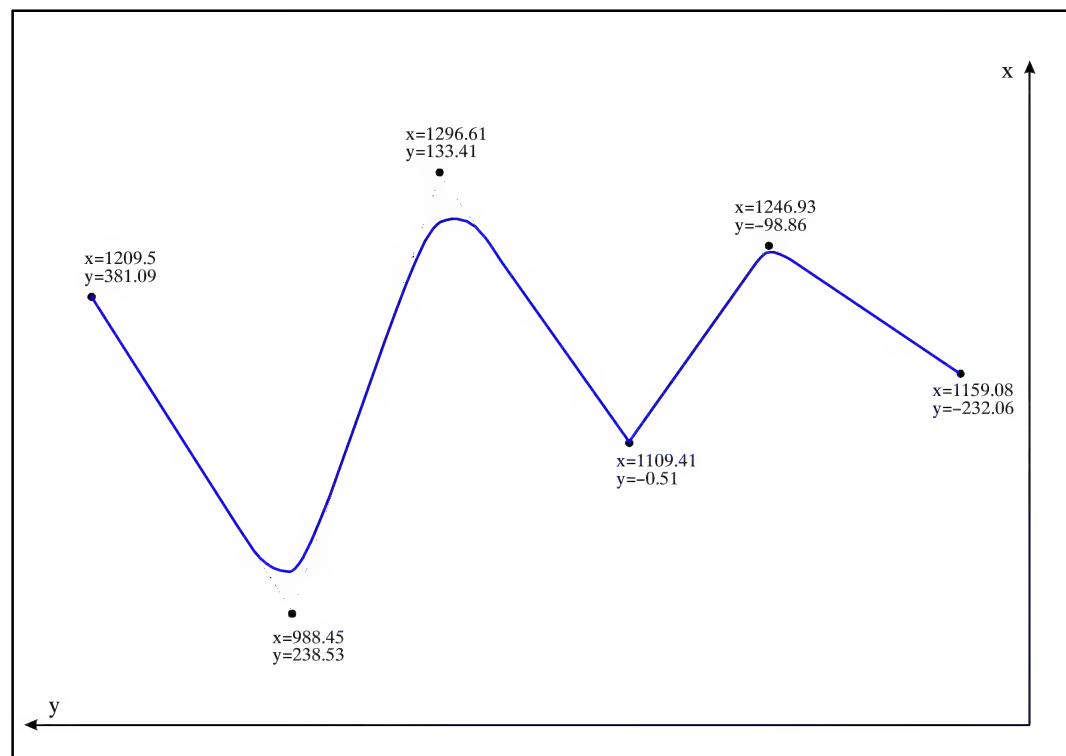


Fig. 30 Example of LIN-LIN approximate positioning



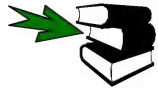
The position at which the approximate positioning contour joins the subsequent LIN block is automatically calculated by the controller. If \$ACC and \$VEL are identical for both individual blocks and the block lengths are sufficient, the approximate positioning parabola will be symmetrical about the bisector between the two individual blocks. With short blocks, the start of approximate positioning is limited to half the length of the block. The velocity is here reduced in such a way that any subsequent exact positioning can always be carried out.

The transitions between individual blocks and the approximate positioning contour are continuous and tangential. This guarantees a “smooth” transition, minimizing mechanical stress, since the velocity components are always continuous.

The contour generated by the controller in the approximate positioning range is independent of override alterations which are permissible at any stage of the motion.

3.5.3 CIRC–CIRC and CIRC–LIN approximate positioning

Approximate positioning between CIRC blocks and other CP blocks (LIN or CIRC) is almost identical to approximate positioning between two LIN blocks. The orientation motion and the translational motion should pass smoothly, from one individual block contour to the next, without sudden changes in velocity. The start of approximate positioning is again defined by the variables \$APO.CDIS, \$APO.CORI or \$APO.CVEL, the evaluation of which is carried out in exactly the same way as for LIN blocks. The desired approximate positioning criterion is again set with the aid of the keywords C_DIS, C_ORI or C_VEL (see 3.5.2).



Further information can be found in the chapter **[Motion programming]**, section **[LIN–LIN approximate positioning]**.

CIRC–CIRC approximate positioning will also be explained with the help of an example and illustrated motion path (see Fig. 31):



```

DEF  UEBERCIR ( );----- Declaration section -----
EXT  BAS (BAS_COMMAND :IN,REAL :IN )
DECL AXIS HOME;----- Initialization -----
BAS (#INITMOV,0 ) ;Initialization of velocities,
                    ;accelerations, $BASE, $TOOL, etc.
HOME={AXIS: A1 0,A2 -90,A3 90,A4 0,A5 0,A6 0};----- Main
section -----
PTP  HOME ; BCO run
PTP  {POS: X 980,Y -238,Z 718,A 133,B 66,C 146,S 6,T 50}
; Space-related variable orientation control
; Approximate positioning using distance criterion
$APO.CDIS=20
CIRC {X 925,Y -285,Z 718},{X 867,Y -192,Z 718,A 155,B 75,C 160}
C_DIS

; Space-related constant orientation control
; End point defined by angle specification
; Approx. pos. not possible because of adv. run stop due to $OUT
$ORI_TYPE=#CONST
CIRC {X 982,Y -221,Z 718,A 50,B 60,C 0},{X 1061,Y -118,Z 718,A
-162,B 60,C 177}, CA 150 C_ORI
$OUT[3]=TRUE

; Path-related variable orientation control
; Approximate positioning using orientation criterion
$ORI_TYPE=#VAR
$CIRC_TYPE=#PATH
CIRC {X 963.08,Y -85.39,Z 718},{X 892.05,Y 67.25,Z 718.01,A
97.34,B 57.07,C 151.11} C_ORI

```

```

; Relative circular motion

; Approximate positioning using velocity criterion
$APO.CVEL=50
CIRC_REL {X -50,Y 50},{X 0,Y 100} C_VEL

; Approximate positioning using distance criterion
$APO.CDIS=40
CIRC_REL {X -50,Y 50},{X 0,Y 100} C_DIS

CIRC_REL {X -50,Y 50},{X 0,Y 100}

PTP HOME
END

```

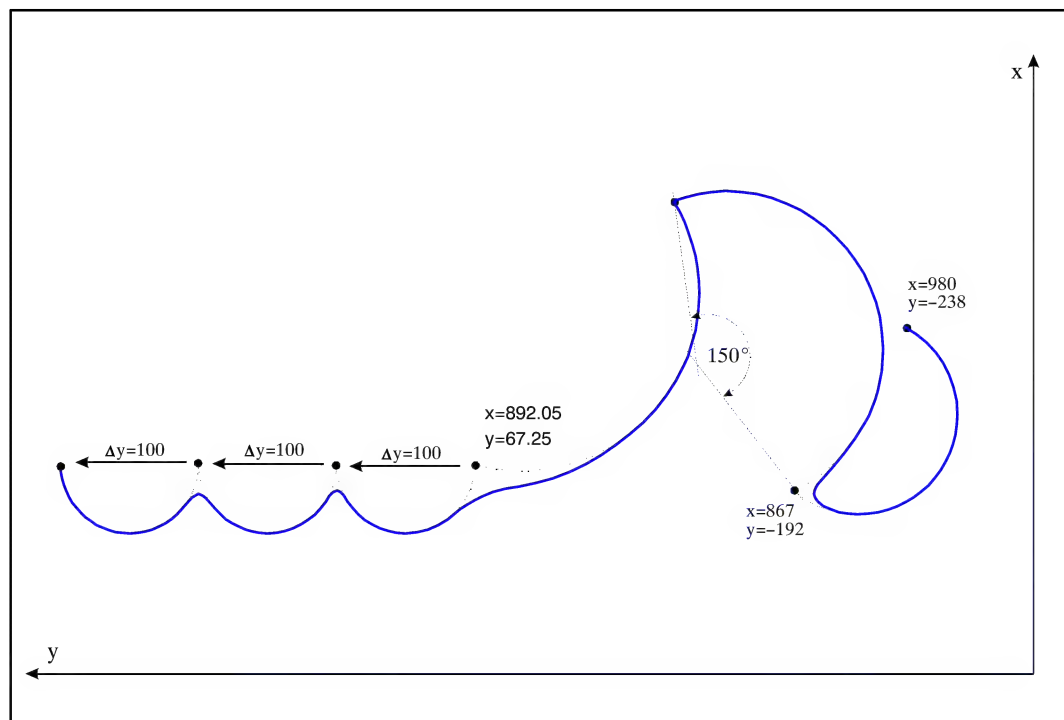


Fig. 31 Example of CIRC-CIRC approximate positioning



In the case of approximate positioning with CIRC blocks, it is not generally possible to calculate a symmetrical approximate positioning contour because of the need for tangential transitions. The approximate positioning path thus consists of two parabolic segments, which have a tangential transition between each other and also to the individual blocks (see Fig. 32).

LIN-CIRC
approximate
positioning

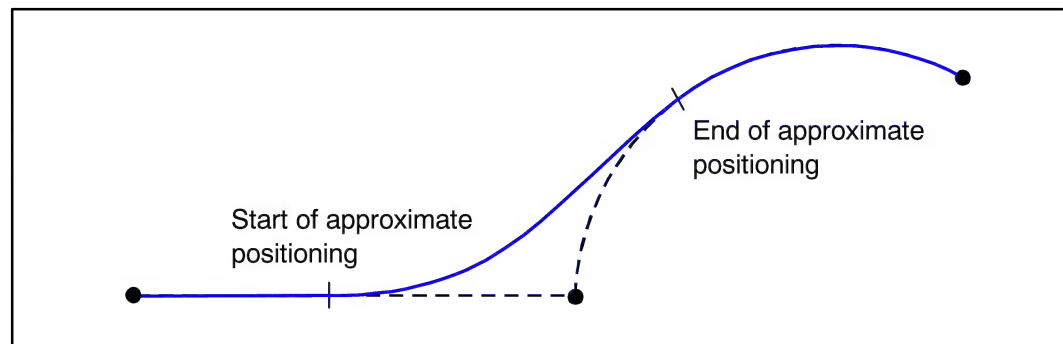


Fig. 32 Approximate positioning contour with LIN-CIRC blocks

In the case of CIRC approximate positioning, interpolation is always space-related. The start orientation is always the orientation reached at the approximate positioning point. If two approximate positioning blocks with path-related orientation are executed, the change in orientation is nonetheless space-related in the approximate positioning range.

3.5.4 PTP→CP approximate positioning

It is possible to approximate axis-specific PTP and Cartesian path motion instructions. The PTP motion offers the following advantages:

- It is fundamentally quicker than its Cartesian counterpart, particularly near singularity positions.
- Unlike Cartesian interpolation, it enables a change of configuration, e.g. a transition from the basic area to the overhead area or a complete swivel through the outstretched wrist position.



The precise path of a PTP motion cannot be predicted exactly as the robot uses the quickest path it can. This path is influenced slightly by a number of factors (e.g. the traversing velocity).

The advantages of axis-specific interpolation can only be used to the full if a continuous transition between axis-specific and Cartesian blocks is possible, because the time won elsewhere is to a large extent lost again in the event of exact positioning.

The programming of PTP→CP approximate positioning is perfectly analogous to the procedures already described. The approximate positioning range is defined in the following manner:

PTP → CP approximate positioning

PTP→CP
approximate
positioning

The beginning of approximate positioning is determined by the PTP criterion \$APO.CPTP in the customary way (see 3.5.1). An approximate positioning criterion (C_DIS, C_ORI, C_VEL), defining entry into the CP block, can be explicitly specified for the following CP motion block.

This is done by means of the instruction sequence:

```
PTP POINT1 C_PTP C_DIS
LIN POINT2
```

If there is no specification in the PTP block for the approximate positioning criterion desired in the CP block, C_DIS is taken as the default value for determining entry into the CP block.



The approximate positioning contour of a PTP→CP or CP→PTP approximation is a PTP motion. It is thus not possible to determine the approximation path exactly.

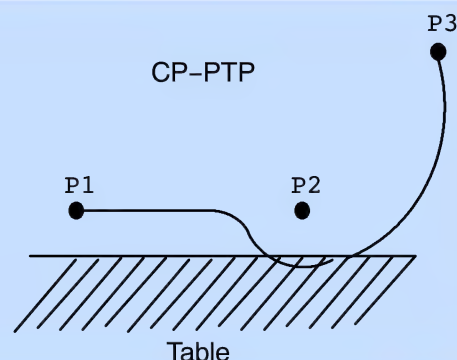
Example:

If a precise path is required, e.g. a CP motion parallel to the table, caution must be exercised in leaving this path by means of a CP→PTP combination with approximate positioning. The same applies to a PTP→CP motion to reach this path.

Program:

```
...
LIN P1
LIN P2 C_DIS
PTP P3
...
```

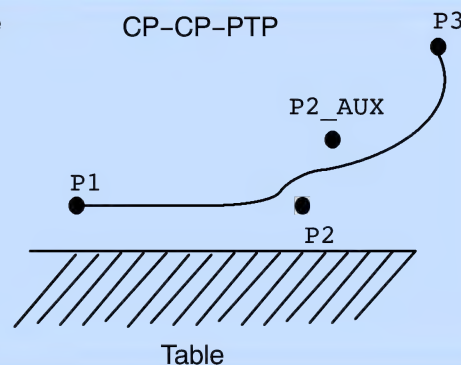
The problem here is that the approximate positioning motion is unpredictable. There is the possibility of a collision with the table.



In order to counter the above problem, while avoiding exact positioning, an additional CP motion (P2_AUX) must be inserted.

Program:

```
...
LIN P1
LIN P2 C_DIS
LIN P2_AUX C_DIS
PTP P3
...
```



CP → PTP approximate positioning

CP→PTP
approximate
positioning

The programmed approximate positioning criterion counts for the CP block, while the system reverts to \$APO.CPTP for the PTP block.

An instruction sequence could thus look like this:

```
CIRC AUX_POINT1 C_VEL
PTP POINT2
```



CP-PTP approximate positioning can only be ensured, however, if none of the robot axes rotates more than 180° in the CP block and if status S remains unchanged, because these position changes cannot be predicted when the approximate positioning contour is planned. If such a change in configuration arises before approximate positioning in the CP block (change in S or T), the path block is executed as an individual block to the programmed end point and the error message "CP/PTP approximation not possible", which must be acknowledged, is displayed. The user should then break up the CP block into several individual blocks, so that the individual block before the CP-PTP approximate positioning is short enough to be able to preclude, for all robot axes, a change in S or T.



Further information can be found in the chapter **[Motion programming]**, section **[Motion commands]**.

In the following example, PTP-LIN, LIN-CIRC and CIRC-PTP approximate positioning have been programmed (see Fig. 33):



```

DEF  UEBERB_P ( )

;----- Declaration section -----
EXT  BAS (BAS_COMMAND :IN,REAL :IN )
DECL AXIS HOME

;----- Initialization -----
BAS (#INITMOV,0 ) ;Initialization of velocities,
;accelerations, $BASE, $TOOL, etc.
HOME={AXIS: A1 0,A2 -90,A3 90,A4 0,A5 0,A6 0}

;----- Main section -----
PTP  HOME ; BCO run
PTP  {POS: X 1281.55,Y -250.02,Z 716,A 79.11,B 68.13,C 79.73,S
6,T 50}

PTP  {POS: X 1209.74,Y -153.44,Z 716,A 79.11,B 68.13,C 79.73,S
6,T 50} C_PTP C_ORI
LIN  {X 1037.81,Y -117.83,Z 716,A 79.11,B 68.13,C 79.73}

$APO.CDIS=25
LIN  {X 1183.15,Y -52.64,Z 716,A 79.11,B 68.13,C 79.73} C_DIS
CIRC {POS: X 1134,Y 53.63,Z 716},{X 1019.21,Y 124.02,Z 716,A
79.11,B 68.12,C 79.73}

CIRC {POS: X 1087.47,Y 218.67,Z 716},{X 1108.78,Y 267.16,Z 716,A
79.11,B 68.12,C 79.73} C_ORI
PTP  {POS: X 1019.31,Y 306.71,Z 716,A 80.8,B 68,C 81.74,S 6,T
59}

PTP  HOME
END

```

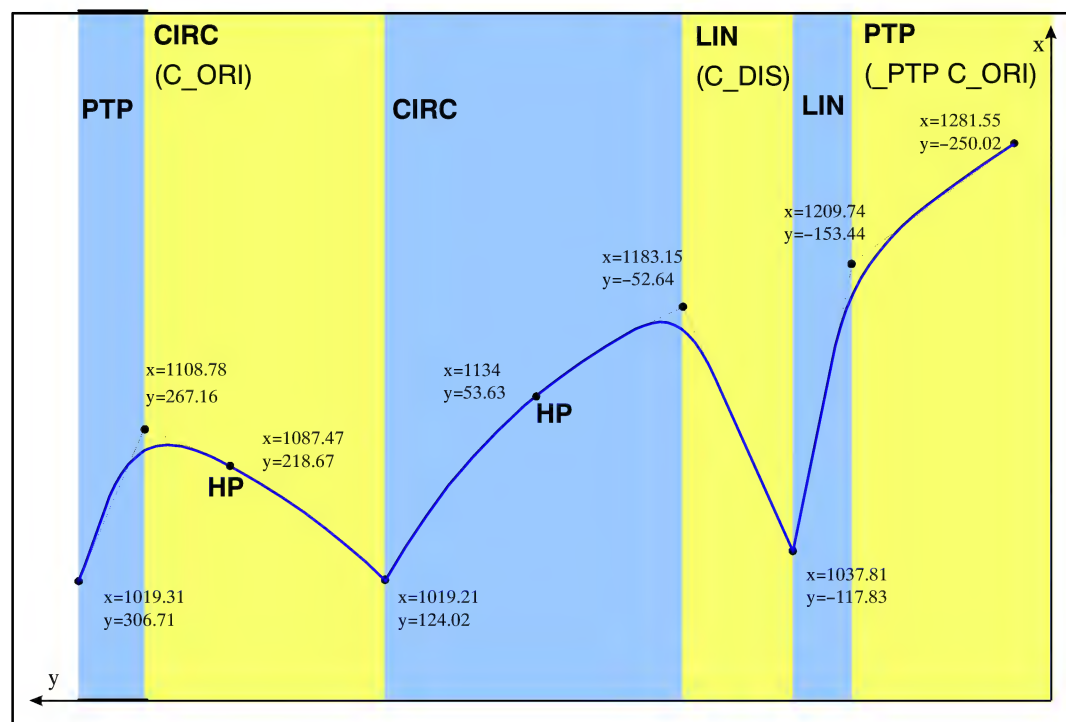


Fig. 33 PTP-CP and CP-CP approximate positioning

3.5.5 Tool change during approximate positioning

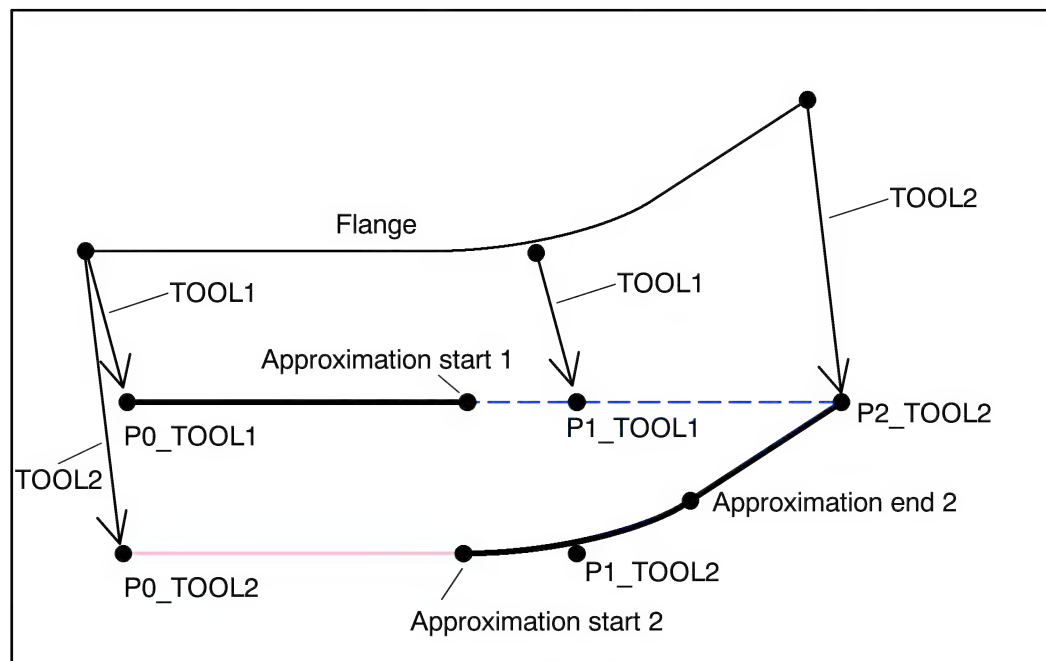
This function is available for all combinations of individual PTP, LIN and CIRC instructions. Even during approximate positioning it is possible to carry out a virtual tool change, i.e. a tool is taught twice and differently each time, e.g. the distance between a spray gun and the workpiece is 5 cm as "TOOL1" and 10 cm as "TOOL2".



Example

In this example, a tool change from TOOL_DATA[1] to TOOL_DATA[2] is desired between points P1 and P2 at the start of the approximate positioning motion.

```
$TOOL=TOOL_DATA[1]
PTP P0_TOOL1;           Point taught with tool TOOL_DATA[1]
LIN P1_TOOL1 C_DIS;     Point taught with tool TOOL_DATA[1]
$TOOL=TOOL_DATA[2]
LIN P2_TOOL2;           Point taught with tool TOOL_DATA[2]
```



The tool change is carried out, in this example, during the approximate positioning, i.e. the Cartesian position jumps at the start of the intermediate block from Approximation start 1 to Approximation start 2; the axis angle and the Cartesian position and velocity of the flange are controlled continuously throughout the entire motion. The jump in the Cartesian TCP position is not eliminated until the motion from Approximation end 2 to P2_TOOL2, so the path follows not the linear course from P1_TOOL1 to P2_TOOL2, but a section of the path programmed for TOOL2 with exact positioning.

3.6 Teaching points

Integration of the teaching procedure is an important quality feature of a robot programming language.

! sign

In KRL you simply program a ! sign as a placeholder for the coordinates to be taught later:

```
PTP !
```

```
LIN ! C_DIS
```

```
CIRC ! ,CA 135.0
```

The relevant robot coordinates can then be saved in the program by pressing the softkey “Change” followed by the softkey “Touch Up”. The current coordinates are written directly into the selected structure in the SRC code, e.g.:

```
PTP {POS:X 145.25,Y 42.46,Z 200.5,A -35.56,B 0.0,C 176.87,S 2,T 2}
```

```
LIN {X -23.55,Y 0.0,Z 713.56,A 0.0,B 34.55,C -90.0} C_DIS
```

```
CIRC {X -56.5,Y -34.5,Z 45.56,A 35.3,B 0.0,C 90.0},{X 3.5,Y 20.30,  
Z 45.56,A 0.0,B 0.0,C 0.0}, CA 135.0
```



When teaching Cartesian coordinates, the base coordinate system (\$BASE) currently valid in the robot system is taken as the reference system. Please always make sure, therefore, when teaching, that the base coordinate system used for the subsequent motion is set.



The KR C... allows for another kind of teaching: program a motion instruction with a variable that you do NOT declare in the declaration section, e.g.:

```
PTP STARTPOINT
```

After pressing the softkeys “Change” and “Var”, you are now prompted to select the desired structure. Once this is done, a `STARTPOINT` variable is automatically declared in the relevant **data list** and assigned the current actual coordinates relative to the current \$BASE, e.g.:

```
DECL FRAME STARTPUNKT={X 15.2,Y 2.46,Z 20.5,A -35.5,B 9.0,C 16.87}
```

If the data list has not been created, the corresponding error message appears.



Further information on data lists can be found in the chapter **[Data lists]**.



As long as you have created a motion instruction by means of the inline forms, you can later use the points taught with the inline form in a KRL motion instruction as well:

The points are stored in the relevant data lists with the name given in the inline form and the prefix X (this is also why a maximum of 11 instead of 12 characters is permissible for point names in inline forms).

The point P7 in the inline form

LN	P7	CONT	Vel=	1.75	m/s	CPDAT1
----	----	------	------	------	-----	--------

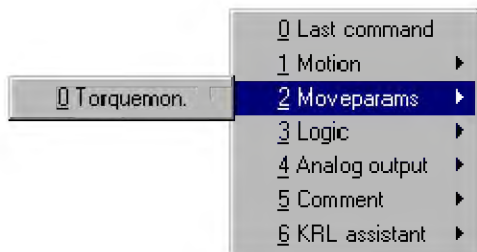
can thus later be addressed as XP7 in a KRL instruction:

```
LIN XP7
```

Bear in mind, here also, that the same base coordinate system must be used in both cases, in order that the robot is positioned to the same point!!

3.7 Motion parameters

This function allows the monitoring tunnel for collision monitoring to be changed. The sensitivity of the collision monitoring can be defined in this way.

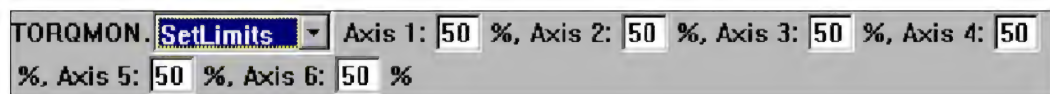


Calling this command opens the following inline form:



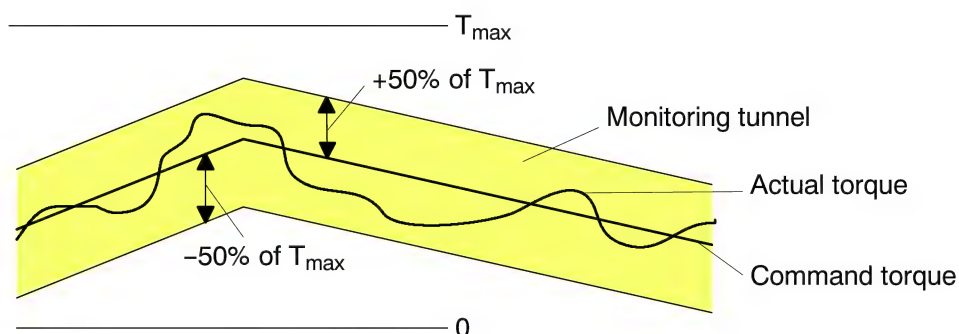
The values defined in the machine data ("C:\Program Files\KRC\MaDa\Steu\\$custom.dat") are used as default values. Alternatively, the user can define different sensitivity levels for different motions and processes.

Alternatively, the user can manually define the amount by which the command torque may deviate from the actual torque. If this is exceeded the robot is stopped.



Example

In the inline form, axes A1...A6 have each been defined with a value of 50%. If the actual torque deviates from the calculated command torque by more than $\pm 50\%$ of the maximum torque, the robot is stopped and a corresponding message is generated in the message window.



The monitoring function does not guarantee that the tool will not be damaged in the event of a collision, but it does reduce the extent of the resulting damage.



More detailed information on collision monitoring can be found in the **Programming Handbook** in the documentation [Configuration], chapter [Configuring the system, Expert].

4 KRL assistant

KUKA technology packages contain the most important functions for normal robot applications. The user can create special functions, which go beyond this scope, by programming the robot system directly in “KRL”, “KUKA Robot Language”.

The “KRL assistant” has been integrated, so that even users who do not often use this programming language can program special functions effectively.

The “KRL assistant” offers the user syntax-supported programming. After the desired KRL command has been selected, instructions relating to the command are offered in masks. The contents of these masks can be either left as they are or changed. All contents can be changed again later as required.

Operator control



To program a motion command, you must select a program or load it in the editor. More detailed information on creating and altering programs can be found in the chapter **[General information on KRL programs]**, section **[Creating and editing programs]**.



```

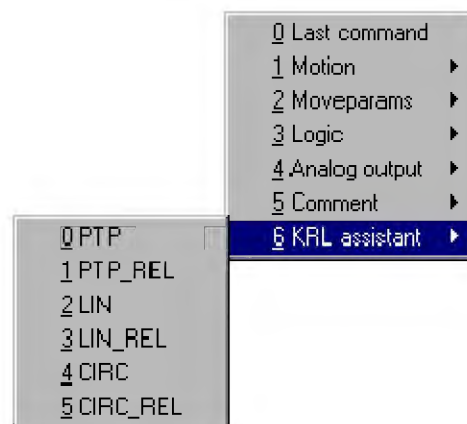
1  → INI
2  PTP HOME Vel= 100 % DEFAULT
3  |
4  PTP HOME Vel= 100 % DEFAULT
5  END

```

Pay attention to the position of the edit cursor. The program line created by you will be inserted as a new line under the cursor.

Commands

Use the menu key “Commands” to open the menu and select the menu item “KRL assistant”. The following submenu is displayed:



You can now make your selection from the motion instructions offered.

4.1 Position specifications

The placeholder “!”



The “!” sign is a placeholder. Using it, it is possible to create a motion program without knowing the exact position of the points which later determine the path of the robot.

When the program is run later, it will stop here and you can teach the point as described below:

Ack

If, during subsequent execution of the program, the message “Instruction not allowed” appears in the message window, delete it using the softkey “Ack”.

Ti...	Nr.	Src.	Message
15:42	1425		Instruction not allowed

Move the robot system to the desired position.

Touch Up

Then press the softkey “Touch Up”. Read the message displayed in the message window.

Confirmation that the position has been saved is displayed in the message window.

Ti...	Nr.	Src.	Message
15:44	0	TPEXPERT	Actual position “[POS: X 1620, Y 0, Z 1910, A 0, B 90, C 0, S 2, T 2]” was trans

Ack

This message can then also be deleted by pressing the softkey “Ack”.

Position specification using variables



Instead of the placeholder you can also enter a valid variable name. A list of keywords reserved for KRL, which you thus cannot use, can be found in [KRL Reference Guide].



You should move the robot system to the desired position before programming this function.



Information on moving the robot manually can be found in the **Operating Handbook** in the documentation **Operator Control**, chapter **[Manuel traversing of the robot]**.

VAR

If the name you have entered is not recognized by the system, the softkey “VAR” appears in the softkey bar. You are prompted to use this softkey to assign a data format to the name.

After it is pressed, the softkey bar changes:

<<	E6POS	POS	FRAME	E6AXIS	AXIS	!
----	-------	-----	-------	--------	------	---



The softkey bar is only available if, alongside the ".SRC" file, a ".DAT" data list also exists.

After one of the softkeys "E6POS", "POS", "FRAME", "E6AXIS" or "AXIS" is pressed, the current position of the robot system is saved under the selected data format. This is confirmed by a message in the message window.

Ti...	Nr.	Src.	Message
15:42	1425		Point KUKA 01 touched with current coordinates

Ack

This message can then be deleted by pressing the softkey "Ack".

Saving the position with "Touch Up"



Before you can save positions in a program using "Touch Up", data regarding the position of the valid robot coordinate system, and also valid tool/workpiece data, must be passed on to the controller.

For this purpose, the INI sequence at the start of the program must be run.



You should move the robot system to the desired position before programming this function.

Touch Up

If you press the softkey "Touch Up", the current position of the robot system is saved.

PTP[{POS: X 1620, Y 0, Z 1910, A 0, B 90, C 0, S 2, T 2}]

A message is displayed in the message window confirming that the position has been saved.

Ti...	Nr.	Src.	Message
15:44	0	TPEXPERT	Actual position "{POS: X 1620, Y 0, Z 1910, A 0, B 90, C 0, S 2, T 2}" was trans

Ack

This message can then be deleted by pressing the softkey "Ack".

Manual position specification

As well as being able to save positions already reached by the robot, you can also manually enter points in space.

{ ? }

To do so, press the softkey "{ ? }" after the inline form has appeared. The assignment of the softkey bar changes:

<<	E6POS	POS	FRAME	E6AXIS	AXIS	!
----	-------	-----	-------	--------	------	---

After a data format has been selected by pressing the corresponding softkey, the current position is saved in the inline form.


```
PTP{A1 0, A2 -90, A3 90, A4 0, A5 0, A6 0}
```

Using the edit functions you can alter these position specifications to suit your requirements.

The geometric operator “:”

Position specifications of types POS and FRAME are combined using the geometric operator “:”. This is always necessary if, for example, the origin of a coordinate system needs to be shifted using a correction value.



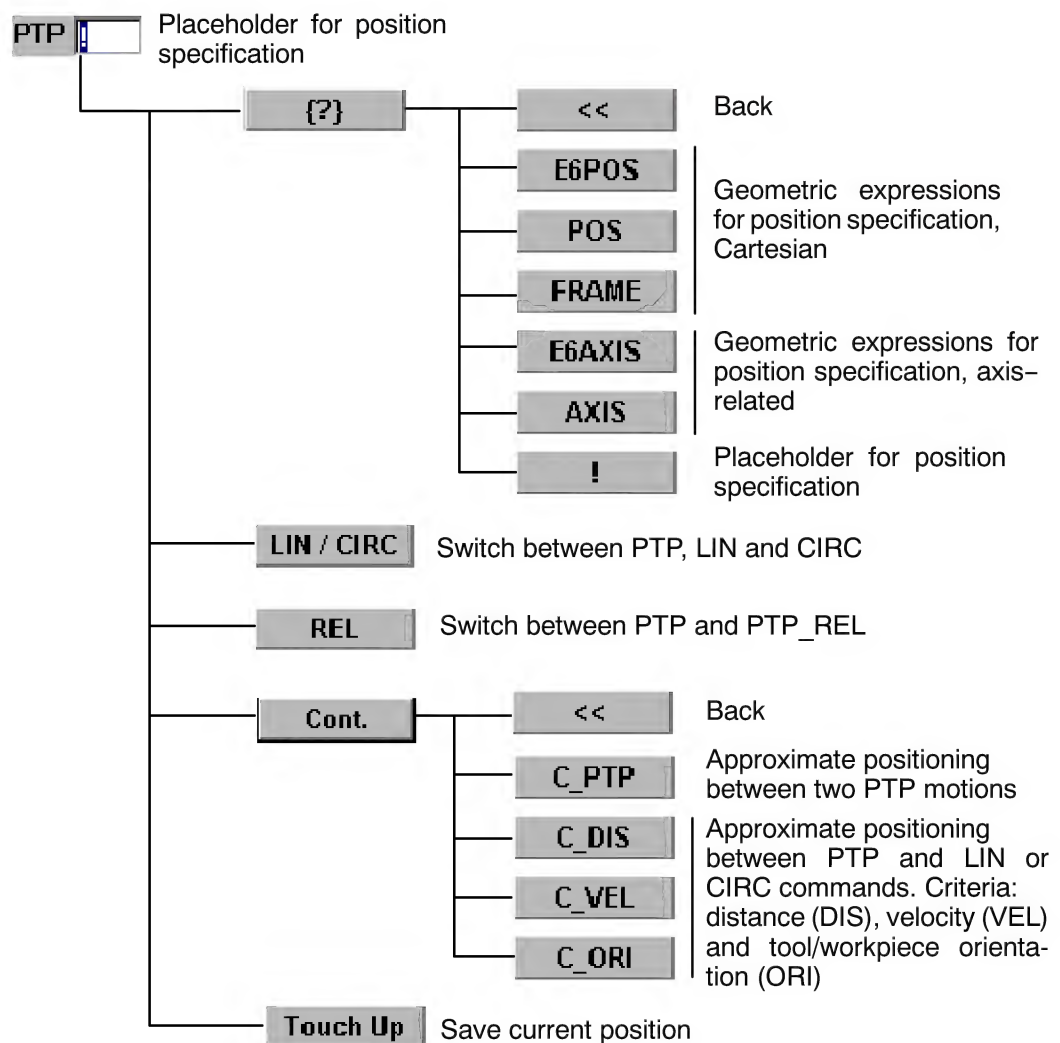
Further information on this can be found in the chapter **[Variables and declarations]**, section **[Data manipulation]** under “**Geometric operator**”.

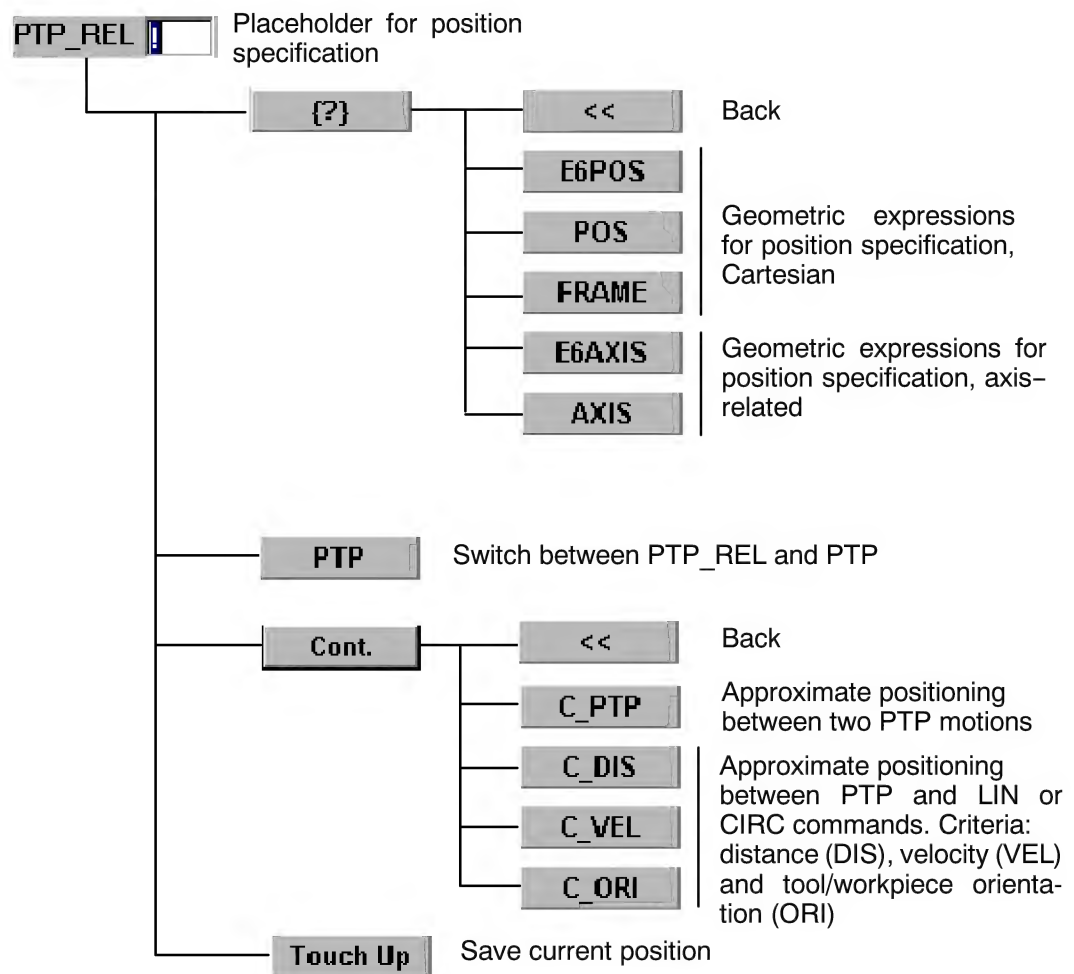
4.2 [PTP] positioning

Here, the robot system is positioned using the quickest route between two points in the work envelope. Since the motion starts and ends in all of the axes at the same time, the axes must be synchronized. The path taken by the robot cannot, therefore, be predicted exactly.



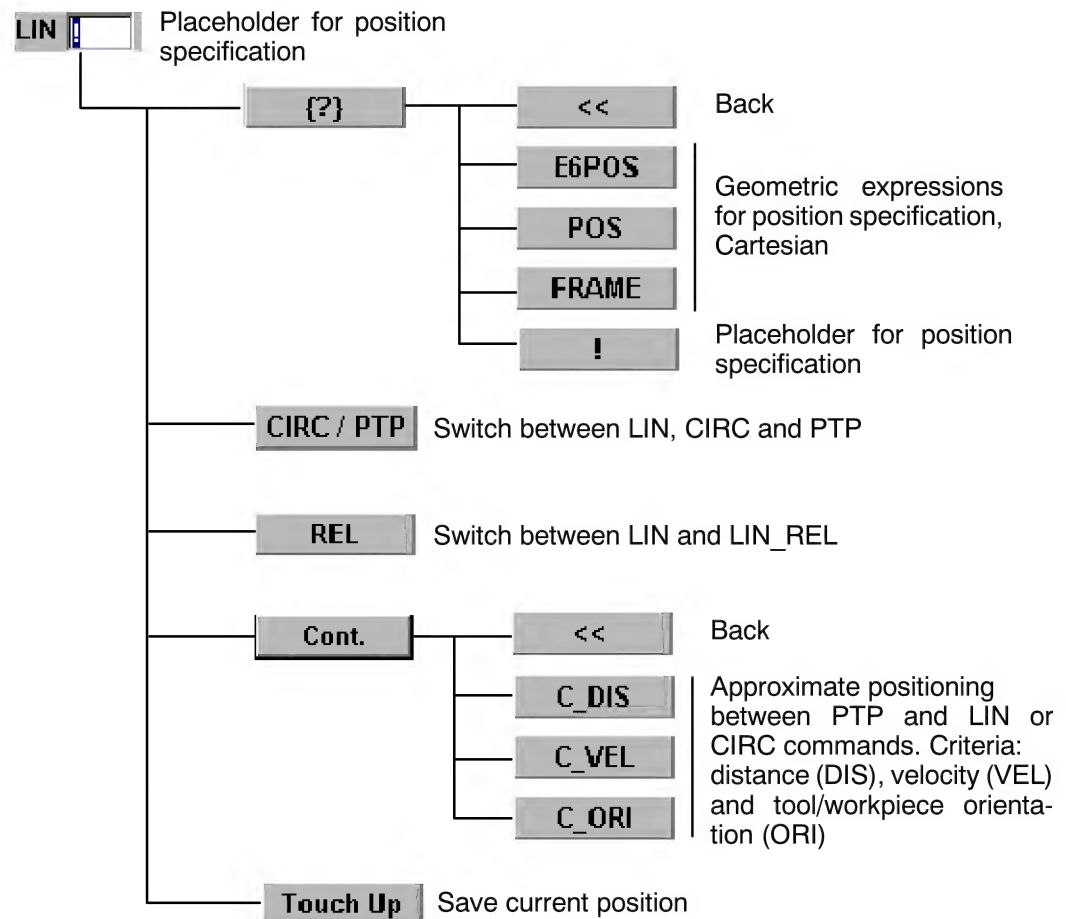
When this instruction is used, the path taken by the robot cannot be predicted exactly. For this reason there is the risk of collision in the proximity of obstacles within the work envelope. The motion characteristics of the robot near obstacles must be tested at reduced velocity!

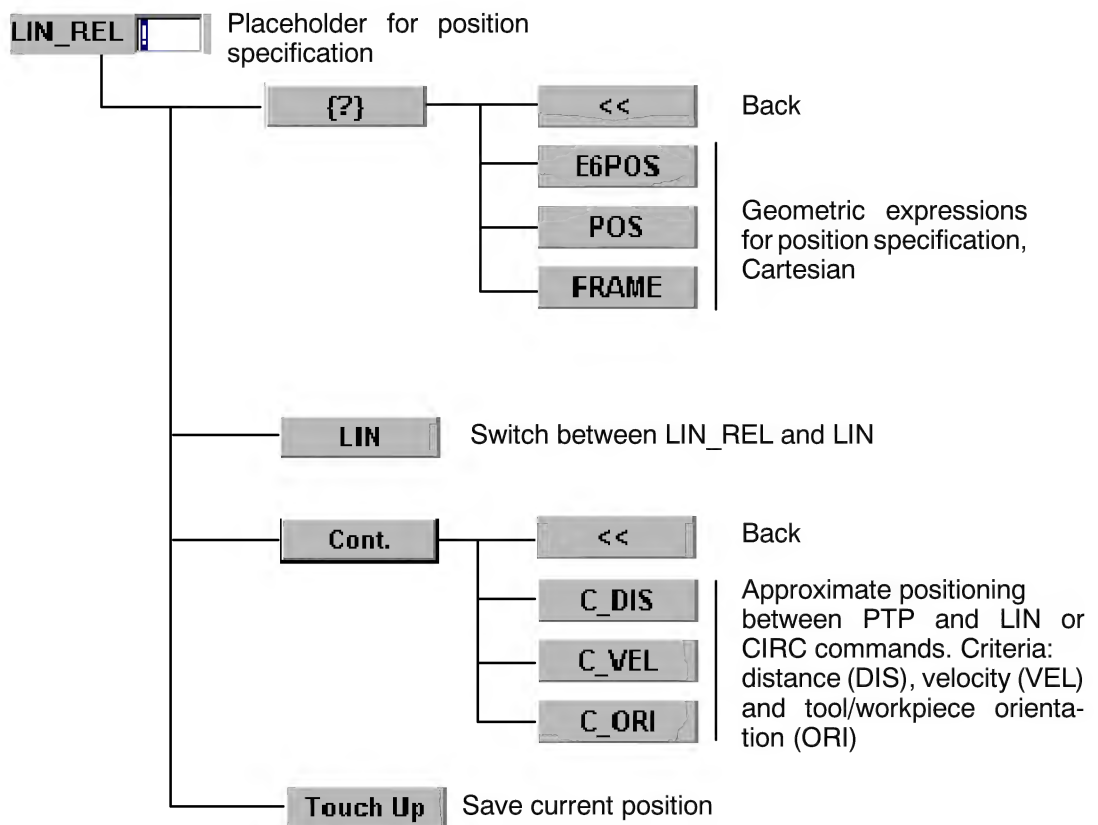




4.3 [LIN] linear motion

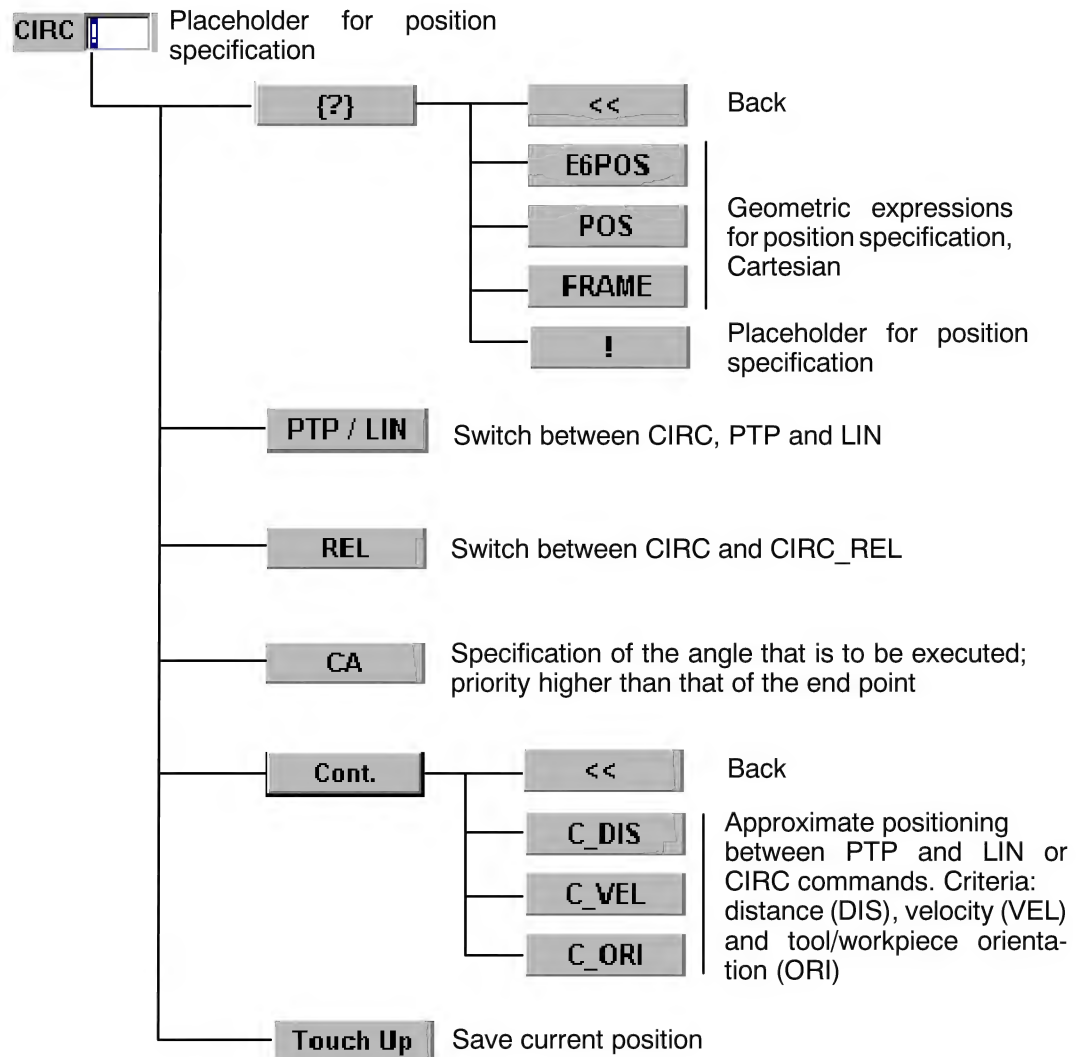
Here, the robot system is positioned using the shortest route between two points in the work envelope – a straight line. The axes of the robot system are here synchronized in such a way that the path velocity remains constant along the length of this straight line.

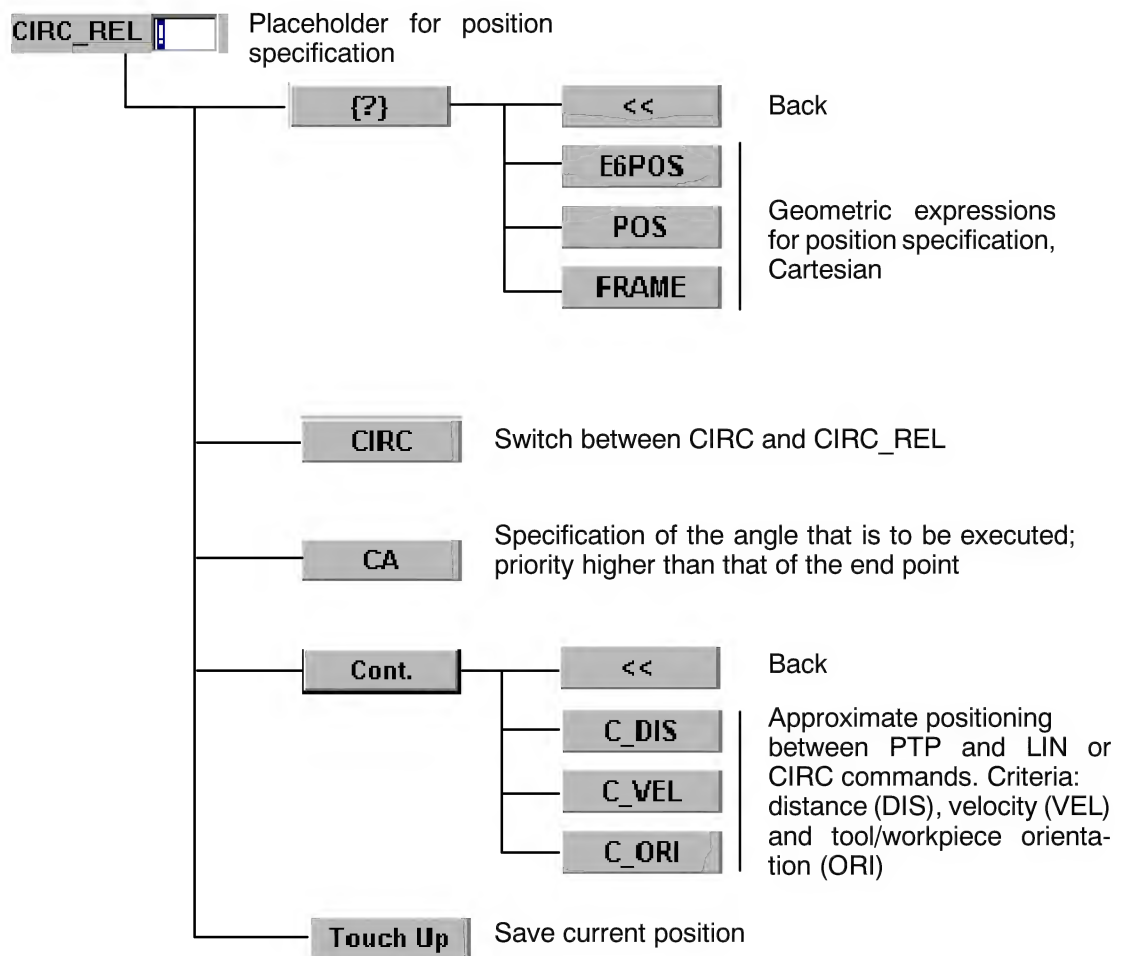




4.4 [CIRC] circular motion

The robot system is here positioned along a circular path in the work envelope defined by start point, auxiliary point and end point. The axes of the robot system are here synchronized in such a way that the path velocity remains constant along this circular path.





5 Program execution control

5.1 Program branches

5.1.1 Jump instruction

The simplest form of program branch is the unconditional GOTO command. This is executed in every case without having to fulfill any conditions. By means of the statement

GOTO

```
GOTO MERKER
```

the program pointer moves to the position MERKER. However, this position must also be defined using the format

MERKER:

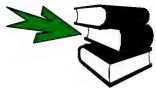
somewhere in the program. The GOTO statement itself does not allow any conclusions to be drawn about the program structure thus created. The name of the GOTO mark should therefore be chosen so as to indicate more clearly the jump it entails. It thus makes a difference whether you write, for example,

```
GOTO MARK_1
```

or

```
GOTO GLUESTOP
```

Since GOTO statements very quickly lead to a loss of structure and clarity within a program, and since, moreover, every GOTO statement can be replaced by a different loop instruction, GOTO statements should be used as little as possible.



An example for “GOTO” can be found in this chapter, section **[Loops]**, under “Non-rejecting loop”.

5.1.2 Conditional branch

IF

The structured IF statement allows instructions to be formulated conditionally with a choice of two alternatives. The general form for these instructions is:

```
IF Execution condition THEN
    Instructions
ELSE
    Instructions
ENDIF
```

The execution condition is a boolean expression. If the execution condition is fulfilled, the THEN block is executed. If it is not fulfilled, the ELSE block can be either executed or dispensed with. If it is dispensed with, the branch is left immediately.

An unlimited number of instructions can be used. In particular, further IF statements can also be used. Nesting of IF blocks is thus possible. Each IF statement must, however, be concluded with its own ENDIF.

In the following program sequence, the robot will move to the HOME position if input 10 is FALSE. If input 10 is set, and if the value of variable A is greater than that of variable B, output 10 is then set and the robot moves to point 1. Irrespective of A and B, if input 1 is not set, the value of variable A is increased by 1 in any case and the robot is moved to the HOME position:

```
...
INT A,B
...
IF $IN[10]==FALSE THEN
    PTP HOME
ELSE
    IF A>B THEN
        $OUT[1]=TRUE
        LIN PUNKT1
    ENDIF
    A=A+1
    PTP HOME
ENDIF
...
```

5.1.3 Switch

SWITCH

Block identifier

If more than 2 alternatives are available, this can either be programmed using a nested IF construction or, much more conveniently, using the SWITCH multi-way branch.

The SWITCH statement is a selection instruction for various program branches. A selection criterion is assigned a certain value ahead of the SWITCH statement. If this value agrees with a block identifier, the corresponding branch is executed and the program jumps straight to the ENDSWITCH statement without taking subsequent block identifiers into consideration. If no block identifier agrees with the selection criterion, the DEFAULT statement block is executed, if there is one. Otherwise, the program resumes at the instruction after the ENDSWITCH statement.

Several block identifiers can be assigned to one program branch. On the other hand, it is not sensible to use one block identifier several times, as only the first branch with the corresponding identifier will ever be taken into consideration.

Permissible data types for the selection criterion are INT, CHAR and ENUM. The data types for the selection criterion and the block identifier must correspond.

The DEFAULT statement can be omitted and may only appear once within a SWITCH statement.

The SWITCH statement can be used, for example, to call up various subprograms by program number. The program number could, for example, be applied to the digital inputs of the KR C... by the PLC (see Section 6.3 for information about the SIGNAL statement). In this way it is available as a selection criterion in the form of an integer value.



```

DEF MAIN()
...
SIGNAL PROG_NR $IN[1] TO $IN[4]
    ;The desired program number is now stored in the
    INT variable PROG_NO by the PLC
...
SWITCH PROG_NO
    CASE 1                ;if PROG_NO=1
        PART_1()
    CASE 2                ;if PROG_NO=2
        PART_2()
        PART_2A()
    CASE 3,4,5            ;if PROG_NO=3, 4 or 5
        $OUT[3]=TRUE
        PART_345()
    DEFAULT              ;if PROG_NO<>1,2,3,4 or 5
        ERROR_UP()
ENDSWITCH
...
END

```



The program CELL (CELL.SRC), available as standard in the controller, is formed in a similar way.

5.2 Loops

The next basic structure for program execution control is the loop; these cause one or more instructions to be repeated until a certain condition is fulfilled. Loops can be distinguished by the form the condition takes, and by the position at which interrogation takes place to see if program execution can be resumed.

A jump into a loop from outside is not allowed and is refused by the controller (error message).

5.2.1 Counting loop

Counting loops are executed until a counting variable either exceeds or falls below a certain end value by counting up or down. The **FOR** statement is available for this in KRL. Using

FOR

```
FOR Counter = Start TO End STEP Increment
    Instructions
ENDFOR
```

a specified number of runs can be very clearly programmed.

Enter integer type expressions as **Start** and **End** values for the counter. The expressions are evaluated once at the start of the loop. The **INT** variable **Counter** (which must be declared in advance) is preset with the start value and then increased or decreased by the programmed increment after each loop execution.

The **increment** must be neither a variable nor zero. If no increment is specified, the default value is 1. Negative increment values are also permissible.

There must be an **ENDFOR** statement for every **FOR** statement. After completion of the last loop execution, the program is resumed with the first instruction after **ENDFOR**.

The counter value can be used either inside or outside the loop. Within the loop, it serves, for example, as an up-to-date index for the processing of arrays. After leaving the loop, the counter retains its final value (i.e. $\text{End} + \text{Increment}$).

In the following example, the axis velocities $\$VEL_AXIS[1] \dots \$VEL_AXIS[6]$ are first set to 100%. The components of a 2-dimensional array are then initialized with the calculated values. The results are shown in Table 20.



```

DEF FOR_PROG()
...
INT I,J
INT ARRAY[10,6]
...
FOR I=1 TO 6
    $VEL_AXIS[I] = 100    ;all axis velocities to 100%
ENDFOR
...
FOR I=1 TO 9 STEP 2
    FOR J=6 TO 1 STEP -1
        ARRAY[I,J] = I*2 + J*J
        ARRAY[I+1,J] = I*2 + I*J
    ENDFOR
ENDFOR
    ;I now has the value 11, J the value 0
...
END

```



Index		I =									
		1	2	3	4	5	6	7	8	9	10
J =	6	38	8	42	24	46	40	50	56	54	72
	5	27	7	31	21	35	35	39	49	43	63
	4	18	6	22	18	26	30	30	42	34	54
	3	11	5	15	15	19	25	23	35	27	45
	2	6	4	10	12	14	20	18	28	22	36
	1	3	3	7	9	11	15	15	21	19	27

Table 20 Result of the calculation in example 5.2

5.2.2 Rejecting loop

WHILE

The WHILE loop requests an execution condition at the start of the repetition. It is a rejecting loop, because it will not run a single time unless the execution condition is satisfied from the outset. The WHILE loop has the following syntax:

```
WHILE Execution condition
    Instructions
ENDWHILE
```

The execution condition is a logical expression which can be a boolean variable, a boolean function call, or a logical operation with a boolean result.

The instruction block is executed if the logic condition has the value TRUE, i.e. the execution condition is fulfilled. If the logic condition has the value FALSE, the program is resumed with the next instruction after ENDWHILE. Each WHILE statement must therefore be ended with an ENDWHILE statement.

The use of WHILE is made clear in example 5.3.



```
DEF WHILE_PR()
...
INT X,W
...
WHILE $IN[4] == TRUE ;Runs as long as input 4 is set
    PTP PALLET
    $OUT[2] = TRUE
    PTP POS_2
    $OUT[2] = FALSE
    PTP HOME
ENDWHILE
...
X = 1
W = 1
WHILE W < 5; ;Runs as long as W is less than 5
    X = X * W
    W = W + 1
ENDWHILE
;W is now 5
;X is now 1•2•3•4 = 24
...
W = 100
WHILE W < 100 ;Runs as long as W is less than 100
    $OUT[15] = TRUE
    W = W + 1
ENDWHILE
... ;Loop never runs, W stays 100
END
```

5.2.3 Non-rejecting loop

REPEAT

The counterpart of the WHILE loop is the REPEAT loop. With REPEAT, the termination condition is not checked until the end of the loop. For this reason, REPEAT loops always run once, even if the termination condition is already fulfilled before the loop begins.

REPEAT

Instructions

UNTIL Termination condition

The termination condition, similarly to the execution condition of a WHILE loop, is a logical expression which can be a boolean variable, a boolean function call, or a logical operation with a boolean result:



```

DEF REPEAT_P()
...
INT W
...
REPEAT
    PTP PALLET
    $OUT[2]=TRUE
    PTP POS_2
    $OUT[2]=FALSE
    PTP HOME
UNTIL $IN[4] == TRUE    ;Runs until input 4 is set
...
X = 1
W = 1
REPEAT
    X = X * W
    W = W + 1
UNTIL W == 4           ;Runs until W equals 4
                        ;W is now 4
                        ;X is now 1•2•3•4 = 24
...
W = 100
REPEAT
    $OUT[15] = TRUE
    W = W + 1
UNTIL W > 100          ;Runs until W is greater than 100
                        ;at least one loop execution, i.e.
                        ;W is now 101, output 15 is set
...
END

```

With WHILE and REPEAT, you now have a very powerful tool for structured programming at your disposal with which you can replace most GOTO commands. The instruction sequence

```
...
X = 0
G = 0
MERKER:
X = X + G
G = G + 1
    IF G > 100 THEN
        GOTO READY
    ENDIF
GOTO MERKER:
READY:
...
```

for example, can be implemented in a much more elegant way using REPEAT:

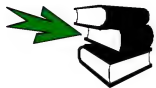
```
...
X = 0
G = 0
REPEAT
    X = X + G
    G = G + 1
UNTIL G > 100
...
```

5.2.4 Endless loop

LOOP You can program endless loops using the **LOOP** statement:

```
LOOP
    Instructions
ENDLOOP
```

The repeated execution of the instruction block can only be terminated using the **EXIT** statement.



Further information on the **Exit instruction** can be found in the next section.

5.2.5 Premature termination of loop execution

EXIT Any loop can be terminated prematurely by using the **EXIT** statement. By calling **EXIT** within the instruction block of a loop, the loop run is immediately terminated and the program resumed after the loop end statement.

Skillful selection of termination and execution conditions renders the **EXIT** statement mostly unnecessary in **REPEAT** and **WHILE** loops. With the endless loop, however, **EXIT** offers the only possibility for terminating execution of the loop. The following example illustrates this:



```
DEF EXIT_PRO()
PTP HOME
LOOP                                ;Start of the endless loop
    PTP POS_1
    LIN POS_2
    IF $IN[1] == TRUE THEN
        EXIT                        ;Terminate when input 1 set
    ENDIF
    CIRC HELP_1,POS_3
    PTP POS_4
ENDLOOP                            ;End of the endless loop
PTP HOME
END
```

5.3 Wait instructions

WAIT Using the **WAIT** statement, you can cause the program to stop until a certain situation arises. A distinction is made between waiting for the occurrence of a certain event and the insertion of wait times.

5.3.1 Waiting for an event

Using the statement

```
WAIT FOR condition
```

you can stop program execution until the event specified under `condition` arises:

- If the logical expression `condition` is already **TRUE** when **WAIT** is called, program execution is not stopped (an advance run stop is triggered, however).
- If `condition` is **FALSE**, program execution is stopped until the expression takes the value **TRUE**.

Examples:

```
WAIT FOR $IN[14]           ;waits until input 14 is TRUE
WAIT FOR BIT_1 == FALSE    ;waits until variable BIT_1 = FALSE
```

If, due to incorrect formulation, the expression can never take the value **TRUE**, the compiler does not recognize this. In this case, program execution is permanently stopped, because the interpreter is waiting for fulfillment of a condition that cannot be satisfied.

5.3.2 Wait times

The **WAIT SEC** statement allows wait times to be programmed in seconds:

```
WAIT SEC time
```

`Time` is an arithmetic **REAL** expression which can be used to specify the number of seconds for which program execution is to be interrupted. If the value is negative, the program does not wait.

Examples:

```
WAIT SEC 17.542
WAIT SEC TIME*4+1
```

5.4 Stopping the program

HALT

If you want to interrupt program execution and stop processing, program the instruction

HALT

The last motion instruction to be executed will, however, be completed. Program execution may only be restarted by pressing the Start key. The next instruction after HALT is then executed.



Special case: In the event of a HALT instruction in an interrupt routine, program execution is only stopped after the advance run has been completely executed (see chapter 8 “Interrupt handling”).

Exception: If a BRAKE instruction is programmed, the robot is stopped immediately.

5.5 Confirming messages

CONFIRM Using the instruction

CONFIRM V_Number

you can confirm acknowledgeable messages under program control. After successful confirmation (acknowledgement), the message specified with the administration number V_Number is no longer available.

After cancelling a stop signal, for example, an acknowledgement message is always generated. This must be acknowledged first before you can work any further. The following subprogram identifies and acknowledges this message automatically, as long as the right operating mode (not manual mode) is selected and the stop status really has been cancelled (since a robot program cannot be started if an acknowledgement message is present, the subprogram has to run in a submit file).



```

DEF AUTO_CONF( )
INT M
DECL STOPMESS MLD ;Predefined structure type for Stop messages
IF $STOPMESS AND $EXT THEN ;Check Stop message and mode
M=MBX_REC($STOPMB_ID,MLD) ;Read current status in MLD
IF M==0 THEN ;Check whether confirmation allowed
IF ((MLD.GRO==2) AND (MLD.STATE==1)) THEN
CONFIRM MLD.CONFNO ;Confirmation of this message
ENDIF
ENDIF
ENDIF
END
  
```

6 Input/output instructions

6.1 General

The KR C... recognizes 1026 inputs and 1024 outputs. In the standard KUKA control cabinet, the following inputs and outputs are available to the user at the X11 connector (MFC module):

Inputs	1 ...16	
Outputs	1 ...16	(with max. capacity 100 mA; 100% simultaneity)
Outputs	17 ...20	(with max. capacity 2 A; 100% simultaneity)

Other inputs/outputs can optionally be configured, using field buses for example.

Inputs can be read, outputs read and written. They are addressed by means of the system variable `$IN[No]` or `$OUT[No]`. Unused outputs can be used as flags.

The inputs/outputs of the MFC module can be reassigned to other areas in the file "IOSYS.INI".



The controller type "KR C2" is supplied as standard without I/Os.



For safety reasons, all input/output instructions and access to inputs/outputs via system variables trigger an advance run stop.

Accessing input/output system variables preceded by a `CONTINUE` instruction does not trigger an advance run stop.



Further information can be found in the chapter **[Motion programming]**, section **[Computer advance run]**.

6.2 Binary inputs/outputs

If inputs/outputs are addressed individually, they are referred to as binary inputs/outputs. Binary outputs can only have 2 states: Low or High. They are therefore treated as `BOOL`-type variables.

With the following system variables, outputs can be

```
$OUT[No] = TRUE      set or
$OUT[No] = FALSE     reset.
```

The state of an `$IN[No]` input can be read into a boolean variable or used as a boolean expression in program execution, interrupt or trigger instructions.



Further information can be found in the chapters **[Program execution control]**, **[Interrupt handling]**, and **[Trigger – Path-related switching actions]**.

The instruction sequences

```
BOOL SWITCH
:
SWITCH = $IN[6]
IF SWITCH == FALSE THEN
:
ENDIF
```

and

```
IF $IN[6] == FALSE THEN
:
ENDIF
```

thus have the same meaning.

SIGNAL

It is also possible, in the KR C..., to assign names to individual inputs or outputs. The signal declaration is used for this purpose. This, as indeed all declarations, must be situated in the declaration section of the program. The following:

```
SIGNAL BREAKER $IN[6]
:
IF BREAKER == FALSE THEN
:
ENDIF
```

can thus also be programmed. The variable Breaker is again internally declared as `BOOL`.



System inputs and outputs can also be addressed using `$IN` and `$OUT`. System outputs, however, are write-protected. Input 1025 is always `TRUE`, input 1026 is always `FALSE`. These inputs are used in the machine data, for example, as “dummy” variables. Repeated use is permissible.

How inputs/outputs are used is explained in example 6.1:



```

DEF BINSIG ( )
;----- Declaration section -----
EXT BAS (BAS_COMMAND:IN,REAL:IN )
DECL AXIS HOME
SIGNAL TERMINATE $IN[16]
SIGNAL LEFT $OUT[13]
SIGNAL MIDDLE $OUT[14]
SIGNAL RIGHT $OUT[15]
;----- Initialization -----
BAS (#INITMOV,0 ) ;Initialization of velocities,
                  ;accelerations, $BASE, $TOOL, etc.
HOME={AXIS: A1 0,A2 -90,A3 90,A4 0,A5 0,A6 0}

;----- Main section -----
PTP HOME ;BCO run
LEFT=FALSE
MIDDLE=TRUE ;in the middle position
RIGHT=FALSE

WHILE TERMINATE==FALSE ;terminate if input 16 is set
  IF $IN[1] AND NOT LEFT THEN ;input 1 set
    PTP {A1 45}
    LEFT=TRUE ;in the lefthand position
    MIDDLE=FALSE
    RIGHT=FALSE
  ELSE
    IF $IN[2] AND NOT MIDDLE THEN ;input 2 set
      PTP {A1 0}
      LEFT=FALSE
      MIDDLE=TRUE ;in the middle position
      RIGHT=FALSE
    ELSE
      IF $IN[3] AND NOT RIGHT THEN ;input 3 set
        PTP {A1 -45}
        LEFT=FALSE
        MIDDLE=FALSE
        RIGHT=TRUE ;in the righthand position
      ENDIF
    ENDIF
  ENDIF
ENDWHILE
PTP HOME
END

```

By setting input 1, 2 or 3, the robot can be moved to three different positions. When the robot has reached the desired position, this is shown by setting the corresponding output 13, 14 or 15.

Since these outputs thus always display the current position of the robot, the following check:

```

IF $IN[1] AND NOT $OUT[13] THEN
:
ENDIF

```

can also be used to prevent the robot from trying, each time the While loop is re-run, to move to the position it is already in. The robot thus only moves if an input is set (i.e. instruction to move to the desired position) **and** the relevant output is **not** set (i.e. the robot is not yet in this position) (see Table 21).

By setting input 16, the While loop and the program are terminated.

\$IN [Nr] AND NOT \$OUT[Nr]		\$OUT[Nr]	
		TRUE	FALSE
\$IN[Nr]	TRUE	FALSE	TRUE
	FALSE	FALSE	FALSE

Table 21 Truth table for an “AND NOT” logical operation

6.3 Digital inputs/outputs

6.3.1 Signal declaration

With the signal declaration, it is possible not only to provide individual inputs/outputs with names, but also to group several binary inputs or outputs together with a digital signal. The declaration

```
SIGNAL OUT $OUT[10] TO $OUT[20]
```

can be used, for example, to address the outputs 10 to 20 as an 11-bit word via the variable OUT, declared internally as an **integer**.

The digital output thus declared can be defined using any permissible integer assignment to the variable OUT, e.g.

```
OUT = 35  
OUT = 'B100011'  
OUT = 'H23'
```

- Inputs/outputs must be specified in the signal declaration without gaps and in ascending sequence.
- A maximum of 32 inputs or outputs can be grouped together under one digital signal.
- An output may appear in more than one signal declaration.

If outputs 13 to 15 from example 6.1 are grouped together under the variable POSITION, this results in the following modified program:



```

DEF  BINSIG_D ( )

;----- Declaration section -----
EXT  BAS (BAS_COMMAND:IN,REAL:IN )
DECL AXIS HOME
SIGNAL TERMINATION $IN[16]
SIGNAL POSITION $OUT[13] TO $OUT[15]

;----- Initialization -----
BAS (#INITMOV,0 ) ;Initialization of velocities,
                  ;accelerations, $BASE, $TOOL, etc.
HOME={AXIS: A1 0,A2 -90,A3 90,A4 0,A5 0,A6 0}

;----- Main section -----
PTP  HOME                ;BCO run

POSITION='B010'          ;in the middle position

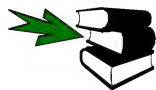
WHILE TERMINATE==FALSE    ;terminate if input 16 is set

  IF $IN[1] AND (POSITION<>'B001') THEN      ;input 1 set
    PTP {A1 45}
    POSITION='B001'                          ;in lefthand position
  ELSE
    IF $IN[2] AND (POSITION<>'B010') THEN      ;input 2 set
      PTP {A1 0}
      POSITION='B010'                        ;in middle position
    ELSE
      IF $IN[3] AND (POSITION<>'B100') THEN;input 3 set
        PTP {A1 -45}
        POSITION='B100'                      ;in righthand position
      ENDIF
    ENDIF
  ENDIF

ENDWHILE

PTP  HOME
END

```



Further information can be found in this chapter, in the section **[Predefined digital inputs]**.

6.3.2 Setting outputs at the end point

As soon as the robot has reached the end point of a motion command, up to 8 outputs can be set, with reference to the main run, and without triggering an advance run stop. The following instruction is used for this:

```
$OUT_C[Output] = Boolean expression
```

Argument	Data type	Meaning
Output	INT	Arithmetical expression defining the number of the output to be set. Outputs 1 ... 1024 are available
Boolean expression	BOOL	Logical expression specifying whether the corresponding output is set to "TRUE" or "FALSE"



Outputs 1 ... 1024 can usually be used. If the variable "\$SET_IO_SIZE" has been set accordingly, 2048 or 4096 outputs are available.

When the advance run pointer reaches the instruction, the Boolean expression is interpreted first. The expression concerned is converted to a constant. The interpreted outputs are then set when the main run pointer reaches this point.

An assignment might look like this:

```
$OUT_C[TOOL[ex]+WELD[y]] = ((NOT(x==100)) AND (SAFE==TRUE))
```

Once interpreting has been completed (advance run pointer), the instruction might then internally look like this:

```
$OUT_C[5] = TRUE
```

When the end point is reached (main run pointer), output 5 is set to the value "TRUE".



When the main run pointer reaches the end point, the output is set to the Boolean value which was valid when the interpreting took place, even if this value has subsequently changed in the meantime.



An "\$OUT_C[x]" instruction, unlike "\$OUT[x]", has no effect on the computer advance run. \$OUT_C[x] can only be written; in order to read an output, "\$OUT[x]" must be used.



Block selection deletes all "\$OUT_C[x]" assignments that have been interpreted, but not yet set. This occurs when block selection is carried out or when a program is reset.

Example PTP, LIN and CIRC

In the case of absolute and relative PTP, LIN and CIRC commands, the output is set immediately after the corresponding motion command:



```
PTP P1
$OUT_C[10]=TRUE
$OUT_C[11]=FALSE
$OUT_C[12]=TRUE
LIN P2
```

If exact positioning is carried out at point “P1”, outputs 10 ... 12 are set here as defined. If, on the other hand, “P1” is approximated, outputs 10 ... 12 are only set when the center of the approximate positioning range has been reached. If approximate positioning is not possible, the outputs are set at point “P1”.



In program run modes “Single-Step” (MSTEP) and “I-Step” (ISTEP), an advance run stop occurs at point P1, but the outputs are not set here. The defined outputs are only set once the Start key has been released and pressed again.



A BCO run also triggers an advance run stop. The outputs are only set as defined once the program has been restarted.

Comparison with trigger

Outputs can be set, with reference to the main run, and without an advance run stop, using either “\$OUT_C[x]” or “Trigger”.



```
$OUT_C[x]:
PTP P1 C_PTP
$OUT_C[5]=TRUE
PTP P2
```

Trigger:

```
TRIGGER WHEN DISTANCE=1 DELAY=0 DO $OUT[5]=TRUE
PTP P1 C_PTP
PTP P2
```

In both cases, output 5 is set as soon as the center of the approximate positioning range of “P1” is reached.

Main run pointer

If the advance run and main run pointers are identical, i.e. the robot is not currently executing a motion command, the assignment is made immediately the “\$OUT_C[]” instruction is reached.



```
PTP P1
WAIT FOR $IN[22]==TRUE
$OUT_C[12]=FALSE
PTP P2
```

In the example, an advance run stop is triggered in the line “WAIT...” if input 22 is “FALSE”. As soon as input 22 has the value “TRUE”, output 12 is set accordingly.

Signal

An output can also be set by means of a 1-bit signal declaration.



```
SIGNAL Test $OUT_C[7]
```

```
PTP P1
```

```
Test = TRUE
```

```
PTP P2
```

In the example, output 7 is set to “TRUE” as soon as the main run pointer has reached the motion command “P1”.

“\$OUT_C[x]” is not permitted in the following situations:

- Within interrupt programs or interrupt declarations
- In a Submit program
- In conjunction with “\$CYCFLAG[x]”
- Within a Trigger command
- In conjunction with the variable correction function

6.4 Pulse outputs

PULSE

Individual outputs can be set or reset for a specified period using the PULSE statement. The instruction

```
PULSE ( $OUT[ 4 ] , TRUE , 0 . 7 )
```

sets output 4, for example, to High level for a period of 0.7 seconds. The pulse can run parallel to the robot program (the interpreter is not stopped).

Instead of direct specification of the output using `$OUT[No]`, a signal variable can also be used.



Further information can be found in this chapter, in the section **[Binary inputs/outputs]**.

Feasible pulse durations lie between 0.012 and 2^{31} seconds. The increment is 0.1 seconds. The controller rounds all values to the nearest tenth of a second.



- A maximum of 16 pulse outputs may be programmed simultaneously.
- High pulses and Low pulses can both be programmed.
- “Program RESET” and “Program CANCEL” both terminate the pulse.
- An active pulse can be influenced by interrupts.
- Pulse outputs can also be programmed at the controller level.
- The PULSE statement triggers an advance run stop. Only in the TRIGGER statement is it executed concurrently with robot motion.



A pulse is NOT terminated by

- an EMERGENCY STOP, operator stop or error-induced stop,
- reaching the end of the program (END statement),
- releasing the Start key if the pulse has been programmed before the first motion instruction and the robot has not yet reached BCO.

In the next program you will find several examples illustrating the use of the PULSE statement:



```

DEF PULSIG ( )

;----- Declaration section -----
EXT BAS (BAS_COMMAND :IN,REAL :IN )
DECL AXIS HOME
INT I
SIGNAL OTTO $OUT[13]

;----- Initialization -----
BAS (#INITMOV,0 ) ;Initialization of velocities,
                  ;accelerations, $BASE, $TOOL, etc.
HOME={AXIS: A1 0,A2 -90,A3 90,A4 0,A5 0,A6 0}

FOR I=1 TO 16
$OUT[I]=FALSE      ;Set all outputs to LOW
ENDFOR

;----- Main section -----
PULSE ($OUT[1],TRUE,2.1) ;Pulse comes direct for 2.1s
PTP HOME                ;BCO run

OTTO=TRUE               ;Set output 13 to TRUE
PTP {A3 45,A5 30}
PULSE (OTTO,FALSE,1.7)  ;LOW pulse for 1.7s at output 13
                        ;Pulse only comes after motion
WAIT SEC 2

FOR I=1 TO 4
PULSE ($OUT[I],TRUE,1)  ;Outputs 1-4 in sequence
WAIT SEC 1              ;for 1s to High
ENDFOR

;Path-related generation of a pulse
TRIGGER WHEN DISTANCE=0 DELAY=50 DO PULSE ($OUT[8],TRUE,1.8)
LIN {X 1391,Y -319,Z 1138,A -33,B -28,C -157}

PTP HOME
CONTINUE               ;Prevent advance run stop for output 15
PULSE ($OUT[15],TRUE,3) ;Pulse comes direct (in the advance
                        ;run) at output 15
PULSE ($OUT[16],TRUE,3) ;Pulse only comes after HOME run
END                    ;and is still set after END

```

In these examples, please note exactly when the programmed pulses are present at the outputs: In principle, the PULSE statement always stops the computer advance run. The pulse is thus not assigned until after completion of the motion.

There are two ways to prevent the advance run stop:

- by programming a CONTINUE statement immediately before the PULSE statement
- by using the PULSE statement in a TRIGGER statement (path-related switching action)



Further information can be found in the chapter **[Motion programming]**, section **[Computer advance run]** (CONTINUE), and in the chapter **[Trigger – path-related switching actions]** (TRIGGER).

6.5 Analog inputs/outputs

Alongside binary inputs/outputs, the KR C... also recognizes analog inputs/outputs. Using optional bus systems, the KR C... makes 8 analog inputs and 16 analog outputs available. Outputs can be read or written using system variables \$ANOUT[1] ... \$ANOUT[16]; inputs can only be read using the variables \$ANIN[1] ... \$ANIN[8].

ANIN
ANOUT

Analog inputs and outputs can be addressed either statically or dynamically, i.e. by continuous polling at the interpolation cycle rate (currently 12 ms). Whereas static reading and writing, as in the case of binary signals, is simply carried out by assigning values, the special instructions ANIN and ANOUT are used for cyclical processing.

6.5.1 Analog outputs

The output values for the 16 analog outputs of the KR C... lie between $-1.0 \dots +1.0$ and are scaled to an output voltage of ± 10.0 V. If the output value exceeds the ± 1.0 limits, the value is cut off.

To set an analog channel a value is simply assigned to the corresponding \$ANOUT variable:

```
$ANOUT[2] = 0.5 ;Analog channel 2 is set to +5 V
```

or

```
REAL V_GLUE  
:
```

```
V_GLUE= -0.9
```

```
$ANOUT[15] = V_GLUE ;Analog channel 15 is set to -9 V
```

These assignments are static because the value of the channel concerned does not change until a new value is explicitly assigned to the relevant \$ANOUT[No] system variable.

It is often desirable, however, for a specific analog output to be continuously recalculated at a defined cycle rate during program execution. This dynamic analog output function is carried out using the ANOUT statement. Using the instruction

```
ANOUT ON WIRE = 0.8 * V_WIRE
```

you can alter the analog output specified with the signal variable WIRE, for example, by simply assigning a value to the variable V_WIRE. The voltage at the corresponding output is determined by the variable V_WIRE.

The variable WIRE must first, of course, be declared in the SIGNAL declaration, e.g.:

```
SIGNAL WIRE $ANOUT[2]
```

Using

```
ANOUT OFF WIRE
```

the cyclical analog output function is terminated again.

The cyclically updated expression, which must be specified for calculating the value of the analog output, must not exceed a certain degree of complexity. The permissible syntax is therefore restricted and technology-oriented. The complete syntax is

ANOUT ON

```
ANOUT ON Signal name = Factor * Control element ± offset <DELAY =t  
<MINIMUM=U1 <MAXIMUM=U2
```

to start the cyclical analog output function, or

ANOUT OFF

```
ANOUT OFF Signal name
```

to end it. The meaning of the individual arguments may be noted from Table 22.

Argument	Data type	Meaning
Signal name	REAL	Signal variable which specifies the analog output (must be declared with <code>SIGNAL</code>). Direct specification of <code>\$ANOUT [No]</code> is not permissible.
Factor	REAL	Any factor, which can be a variable, signal name or constant.
Control element	REAL	The control element affects the analog output. It can be a variable or a signal name.
Offset	REAL	An offset can optionally be programmed as a control element. The offset must be a constant.
t	REAL	By using the keyword <code>DELAY</code> and entering a positive or negative amount of time in seconds, the cyclically calculated output signal can optionally be either delayed (+) or set early (-).
U1	REAL	<p>The keyword <code>MINIMUM</code> defines the minimum voltage present at the output. Permissible values are $-1.0 \dots 1.0$ (corresponding to $-10 \text{ V} \dots +10 \text{ V}$). The minimum value must be less than the maximum value if both values are used.</p> <p>The value may also be a variable, a structure component or an array element.</p>
U2	REAL	<p>The keyword <code>MAXIMUM</code> defines the maximum voltage that may be sent to the output. Permissible values are $-1.0 \dots 1.0$ (corresponding to $-10 \text{ V} \dots +10 \text{ V}$). The maximum value must be greater than the minimum value if both values are used.</p> <p>The value may also be a variable, a structure component or an array element.</p>

Table 22 Arguments in the `ANOUT` statement

The optional parameters “Minimum” and “Maximum” are not available in the inline forms as these are used exclusively for the technology “Adhesive bonding”.



In this example, both a minimum and a maximum value have been defined. The corresponding entries are “MINIMUM=0.3” and “MAXIMUM=0.95”.

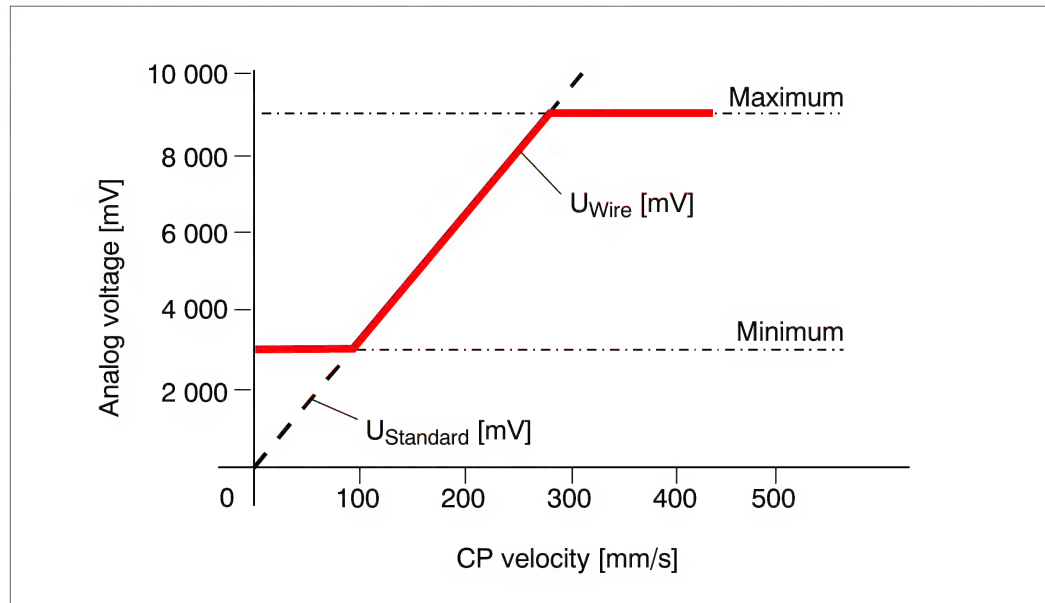


Fig. 34 Analog voltage dependent on the CP velocity

6.5.2 Analog inputs

The 8 analog inputs of the KR C... can be read using the variables \$ANIN[1] to \$ANIN[8] by simply assigning a value to a REAL variable:

```
REAL PART
:
PART = $ANIN[ 3 ]
```

or

```
SIGNAL SENSOR3 $ANIN[ 3 ]
REAL PART
:
PART = SENSOR3
```

The values in \$ANIN[No] range between +1.0 and -1.0 and represent an input voltage of +10 V to -10 V.

ANIN

The ANIN statement is used for the cyclical reading of analog inputs. Using it, up to 3 analog inputs can be read simultaneously. They are read at the interpolation cycle rate.

Using the instruction sequence

```
SIGNAL SENSOR3 $ANIN[ 3 ]
REAL PART
:
ANIN ON PART = 1 * SENSOR3
```

you can thus cyclically read analog input 3, and using the instruction

```
ANIN OFF SENSOR3
```

you can end the reading again.

Bear in mind that a maximum of 3 ANIN ON statements may be active at the same time. It is permissible to access the same analog interface or to define the same variable in both statements.

The complete syntax for cyclically reading an analog input is:

ANIN ON

ANIN ON value = Factor * Signal name <± offset

Using

ANIN OFF

ANIN OFF Signal name

the cyclical monitoring is ended. The meaning of the arguments can be found in Table 23.

Argument	Data type	Meaning
Value	REAL	The value can be a variable or an (output) signal name. The result of the cyclical reading is stored in value.
Signal name	REAL	Signal variable which specifies the analog input (must be declared with SIGNAL). Direct specification of \$ANIN[No] is not permissible.
Factor	REAL	Any factor, which can be a variable, signal name or constant.
Offset	REAL	An offset can optionally be programmed. The offset can be a constant, variable or signal name.

Table 23 Arguments in the ANIN statement

The instructions relating to analog inputs and outputs are illustrated in the following example. By means of the system variable \$TECHIN[1] and a path tracking sensor connected to an analog input, a path correction, for example, can be carried out during the motion. If weighted with the relevant factors, the variable \$VEL_ACT, which always contains the current path velocity, can be used as a velocity-proportional analog output, for example, to control the amount of adhesive dispensed in bonding applications.



```

DEF  ANSIG ( )

;----- Declaration section -----
EXT  BAS (BAS_COMMAND :IN,REAL :IN )
DECL AXIS HOME
INT I
SIGNAL ADHESIVE $ANOUT[1]           ;Open nozzle for adhesive
SIGNAL CORRECTION $ANIN[5]         ;Path tracking sensor

;----- Initialization -----
BAS (#INITMOV,0 ) ;Initialization of velocities,
                    ;accelerations, $BASE, $TOOL, etc.
HOME={AXIS: A1 0,A2 -90,A3 90,A4 0,A5 0,A6 0}

FOR I=1 TO 16
$ANOUT[I]=0           ;Set all outputs to 0V
ENDFOR

;----- Main section -----
PTP HOME                ;BCO run

$ANOUT[3] = 0.7        ;Analog output 3 to 7 V

IF $ANIN[1] >= 0 THEN   ;Adhesive process only if analog
                        ;input 1 has a positive voltage
    PTP POS1

    ;Path correction according to sensor signal with the aid of
    ;system variable $TECHIN
    ANIN ON $TECHIN[1] = 1 * CORRECTION + 0.1

    ; Velocity-proportional analog output;System variable $VEL_ACT
    ;contains the current path velocity
    ANOUT ON ADHESIVE = 0.5 * $VEL_ACT + 0.2 DELAY = -0.12

    LIN POS2
    CIRC INTERMEDPOS,POS3
    ANOUT OFF ADHESIVE
    ANIN OFF CORRECTION

    PTP POS4

ENDIF

PTP HOME
END

```

6.6 Predefined digital inputs

The controller has 6 digital inputs available, which can be read using the signal variables \$DIGIN1...\$DIGIN6. The inputs are included in the normal user inputs. They can be 32 bits long and have a corresponding strobe output.

The inputs are configured in the machine data: "/mada/steu/\$machine.dat". A signal declaration defines, firstly, the range and size of a digital input:

```
SIGNAL $DIGIN3 $IN[1000] TO $IN[1011]
```

Sign interpretations, corresponding strobe outputs and the type of strobe signal are defined using the additional system variables \$DIGIN1CODE...\$DIGIN6CODE, \$STROBE1...\$STROBE6 and \$STROBE1LEV...\$STROBE6LEV:

```
DECL DIGINCODE $DIGIN3CODE = #UNSIGNED ;Not preceded by a sign
```

```
SIGNAL $STROBE3 $OUT[1000] ;Define strobe output
```

```
BOOL $STROBE3LEV = TRUE ;Strobe is a High pulse
```

A strobe output is a KR C... output with a specified pulse which freezes the signal from an external device (e.g. rotary encoder) so that it can be read.

Whereas more than one digital input can access the same input, strobe signals may NOT define the same output.

The range of values for \$DIGIN1...\$DIGIN6 depends on the defined bit length as well as on the sign interpretation (#SIGNED or #UNSIGNED):

12 bits with sign (#SIGNED) Range of values: -2048...2047

12 bits without sign (#UNSIGNED) Range of values: 0...4095

The digital inputs can either be read statically by means of the usual value assignment:

```
INT NUMBER
```

```
:
```

```
NUMBER = $DIGIN2
```

DIGIN

or cyclically using a DIGIN statement:

```
INT NUMBER
```

```
:
```

```
DIGIN ON NUMBER = FACTOR * $DIGIN2 + OFFSET
```

```
:
```

```
DIGIN OFF $DIGIN2
```

A total of 6 DIGIN ON instructions are allowed at the same time. Analog input signals can also be accessed in the DIGIN ON statement (e.g. as FACTOR). The syntax is fully analogous with that for the ANIN ON statement:

DIGIN ON

DIGIN ON value = Factor * Signal name <± Offset

DIGIN OFF

DIGIN OFF Signal name

The meaning of the individual arguments is described in Table 24.

Argument	Data type	Meaning
Value	REAL	The value can be a variable or an (output) signal name. The result of the cyclical reading is stored in <code>value</code> .
Signal name	REAL	Signal variable which specifies the digital input. Only <code>\$DIGIN1...\$DIGIN6</code> are permissible.
Factor	REAL	Any factor, which can be a variable, signal name or constant.
Offset	REAL	An offset can optionally be programmed. The offset can be a constant, variable or signal name.

Table 24 Arguments in the DIGIN statement

7 Subprograms and functions

In order to reduce the amount of typing and the program length when dealing with similar, often repeated program sections, subprograms and functions have been introduced as language constructs.

One effect of subprograms and functions that should not be underestimated with large programs is the possibility of re-using, in other programs, algorithms that have already been written, and in particular the use of subprograms for structuring the program. This structuring process can bring about a hierarchical configuration so that individual subprograms, called up by a higher-level program, can process tasks completely and pass on the results.

7.1 Declaration

A subprogram or function is a separate program section, with its own program descriptor, declaration section and instruction section, which can be called from any position in the main program. After execution of the subprogram or function, the program jumps back to the next command after the subprogram call (see Fig. 35).

Further subprograms and/or functions can be called from within a subprogram or function. The maximum permissible nesting depth is 20. If this is exceeded, the error message "PROGRAM STACK OVERFLOW" is generated. Recurrent calling of subprograms and functions is allowed. In other words, a subprogram or function can recall itself.

DEF

All subprograms are declared in exactly the same way as the main program, with the DEF declaration plus name, and concluded with END, e.g.:

```
DEF UNTERPR()
...
END
```

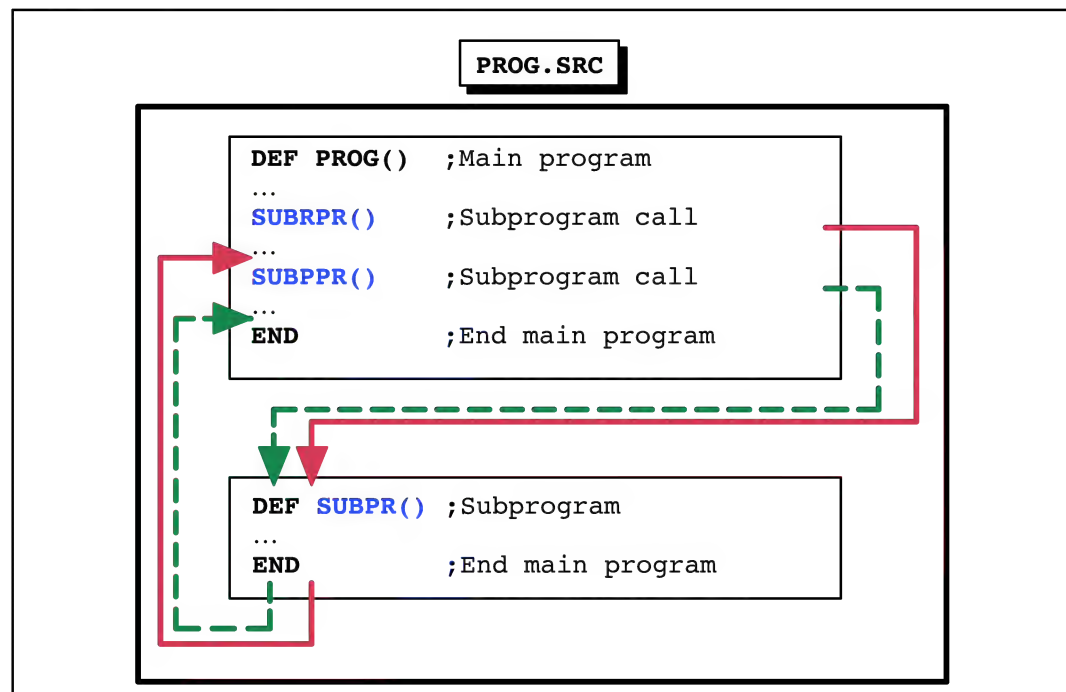


Fig. 35 Subprogram call and return to main program

DEFFCT A function is a type of subprogram; the program name is, at the same time, however, a variable of a specific data type. The result of the function can thus be passed on by simply assigning a value to a variable. For this reason, when declaring functions with the keyword **DEFFCT**, the data type of the function must also be specified along with the name of the function. A function is concluded with **ENDFCT**. Since a function is supposed to communicate a value, this value must be specified using the **RETURN** statement before the **ENDFCT** statement. Example:

```
DEFFCT INT FUNCTION( )
...
RETURN( X )
ENDFCT
```

local A fundamental distinction is made between local and global subprograms or functions. In the case of local subprograms or functions, the main program and the subprograms/functions are found in the same SRC file. The file bears the name of the main program. The main program is always situated at the head of the source text, and can be followed by any quantity of subprograms and functions in any order.

global Local subprograms/functions can only be called from within the SRC file in which they were programmed. If it is necessary to be able to call subprograms/functions from other programs they must be global, i.e. saved in a separate SRC file. Global subprograms or functions are saved in a separate SRC file. In this way, every program becomes a subprogram if it is called by another program (main program, subprogram or function).



- All variables declared in the data list of the main program are recognized in local subprograms and functions. Variables which have been declared in the main program (SRC file) are so-called “runtime variables” and may only be used in the main program. Attempting to use these variables in the subprogram causes a corresponding error message to be generated. Variables declared in the main program are not recognized in global subprograms or functions.
- Variables declared in subprograms or functions are not recognized in the main program.
- A main program cannot access the local subprograms or functions of another main program.
- The maximum length of local subprogram/function names is 24 characters. The maximum length with global subprograms/functions is 20 characters (because of the file extensions).

In order for the global subprogram to be known to the main program, it needs simply to be called in the main program (e.g. **PROG_2 ()**). By specifying the parameter list (see 7.2), the necessary memory space is also unambiguously defined. Examples:

```
PROG_3 ( )
FUNCTION( REAL: IN )
```

The difference between local and global subprograms/functions is illustrated in Fig. 36: **PROG.SRC**, **PROG_1.SRC** and **PROG_3.SRC** are each independent main programs, while **PROG_2FUN.SRC** is a function. Calling a program (e.g. **PROG_1.SRC**) from **PROG.SRC** automatically turns it into a global subprogram. **LOCAL ()** is a local subprogram, **LOCALFUN ()** is a local function of **PROG.SRC**.

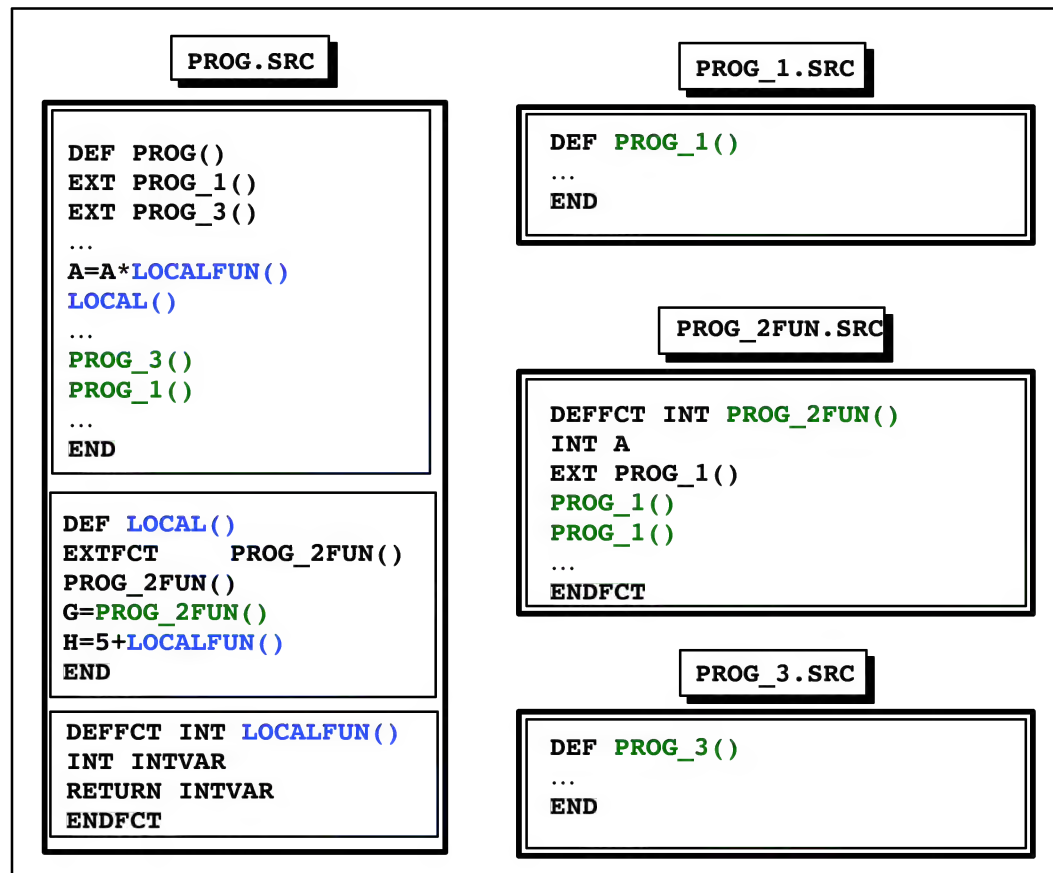


Fig. 36 Difference between local and global subprograms

7.2 Subprogram and function call and parameter transfer

A subprogram is called by entering the subprogram name plus round brackets. In this way it looks like an instruction (see Section 1.1), e.g.:

```
SUBPROG1 ( )
```

A function call is a special kind of value assignment. A function can thus never stand alone; instead, the function value must constantly be assigned within the framework of a variable expression of the same data type, e.g.:

```
INTVAR = 5 * INTFUNCTION( ) + 1
REALVAR = REALFUNCTION( )
```

Parameter list

All variables declared in the data list of the main program are recognized in local subprograms and functions. In global subprograms, on the other hand, these variables are not recognized. Values can also be transferred to global subprograms and functions, however, using a parameter list.

Transfer using parameter lists is also often useful in local subprograms and functions, as a clear distinction can be made in this way between the main program and the subprogram/function. All variables declared in the main program (SRC file) are only used there; all transfers to subprograms and functions (local and global) are carried out using parameter lists. Programming in this structured way significantly reduces programming errors.

There are two different mechanisms for transferring parameters:

- **Call by value (IN)**
With this kind of transfer, a **value** from the main program is transferred to a variable in the subprogram or function. The transferred value can be a constant, a variable, a function call or an expression. Where different data types are involved, type matching is carried out where possible.
- **Call by reference (OUT)**
With "Call by reference", only the **address** of a variable from the main program is transferred to the subprogram or function. The subprogram or function called can now overwrite the memory area using a variable name of its own and in this way also alter the value of the variable in the main program. The data types must therefore be identical; type matching is not possible in this case.

The difference between these two methods is shown in Fig. 37. Whereas, with "Call by value", variable **x** remains unchanged in the main program because of the separate memory areas, in "Call by reference" it is overwritten with the variable **NUMBER** in the function.

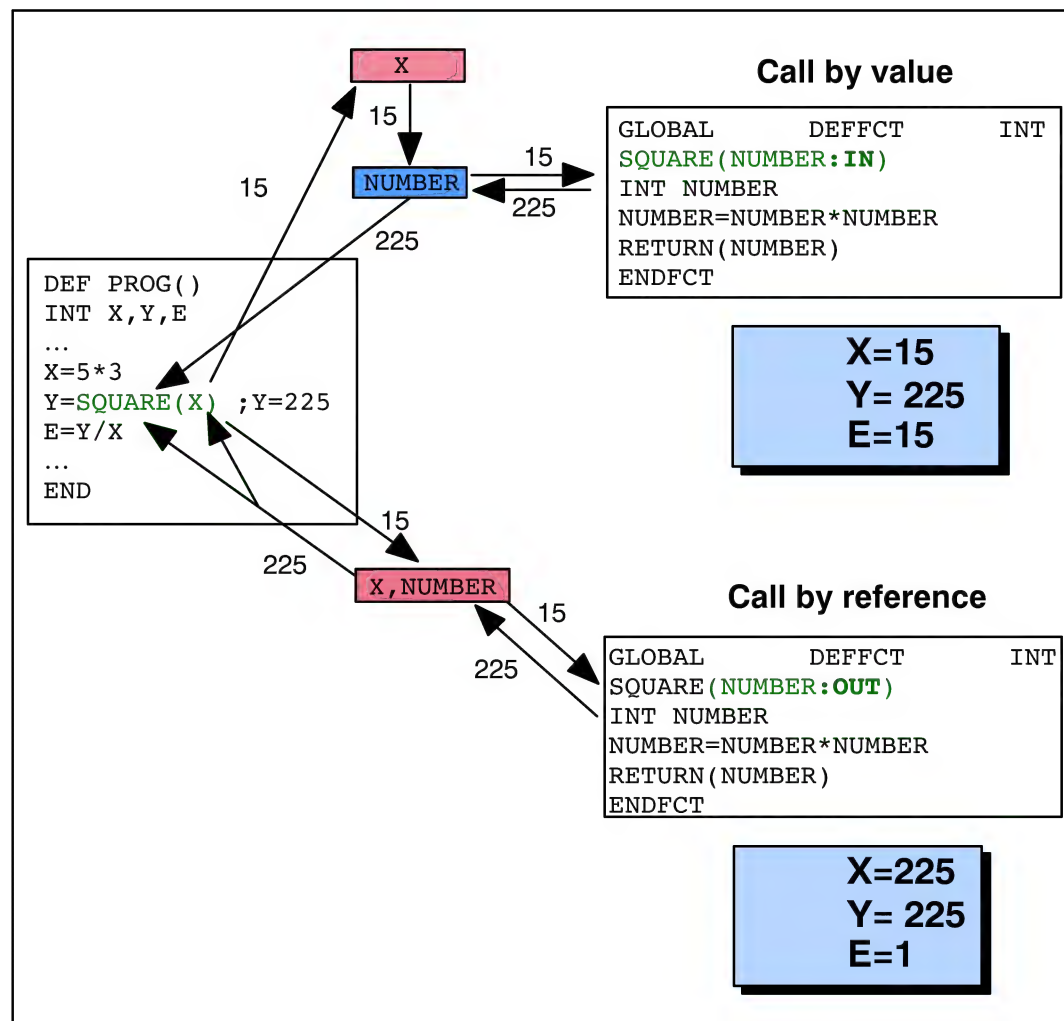


Fig. 37 Difference between "Call by value" and "Call by reference"

"Call by value" is entered in the subprogram or function by the keyword **IN** after every variable in the parameter list. "Call by reference" is obtained by entering the word **OUT**. **OUT** is also the default setting. Example:

```
DEF CALCULATE (X:OUT,Y:IN,Z:IN,B)
```

If the global subprogram or function to be called has not been declared as **GLOBAL**, the external declaration in the main program must specify what data type the respective variables are and which transfer mechanism is to be used. The default setting is again **OUT**. Example:

```
EXTFCT REAL FUNCT1 (REAL:IN,BOOL:OUT,REAL,CHAR:IN)
```

The use of **IN** and **OUT** is illustrated in the following example. The subprogram and the function are global.



```

DEF PROG( )
CALCULATE (INT:OUT,INT:IN,INT:IN)
FUNCT1(REAL:IN,REAL:OUT,REAL:OUT,REAL:IN,REAL:OUT)
INT A,B,C
REAL D,E,F,G,H,X

A = 1
B = 2
C = 3
D = 1
E = 2
F = 3
G = 4
H = 5

CALCULATE (A,B,C)
      ;A is now 11
      ;B is now 2
      ;C is now 3

X = FUNCT1(H,D,E,F,G)
      ;D is now 3
      ;E is now 8
      ;F is now 3
      ;G is now 24
      ;H is now 5
      ;X is now 15
END

DEF CALCULATE(X1:OUT,X2:IN,X3:IN)                                ;Global
SP
INT X1,X2,X3
X1=X1+10
X2=X2+10
X3=X3+10
END

DEFFCT REAL FUNCT1(X1:IN,X2:OUT,X3:OUT,X4:IN,X5:OUT);Global fct.
REAL X1,X2,X3,X4,X5
X1 = X1*2
X2 = X2*3
X3 = X3*4
X4 = X4*5
X5 = X5*6
RETURN(X4)
ENDFCT

```

When transferring an array, the array in the subprogram or function must also be declared again, but without an index. For this, please refer to the following example in which the values of an array X[] are doubled (the function is global):



```

DEF  ARRAY ( )
EXT BAS (BAS_COMMAND:IN,REAL:IN)
INT X[5]      ;Array declaration
INT I

BAS (#INITMOV,0)

FOR I=1 TO 5
  X[I]=I      ;Initialize array X[]
ENDFOR      ;X[1]=1,X[2]=2,X[3]=3,X[4]=4,x[5]=5

DOUBLE (X[]) ;Call subprogram with array parameters
          ;X[1]=2,X[2]=4,X[3]=6,X[4]=8,X[5]=10
END

DEF  DOUBLE (A[]:OUT)
INT A[]      ;Repeat declaration of the array
INT I
FOR I=1 TO 5
  A[I]=2*A[I] ;Doubling of the array values
ENDFOR
END

```

Similarly, no indices are specified when transferring multi-dimensional arrays; however, the dimensions of the array must be specified by entering commas. Examples:

A[,] for two-dimensional arrays
 A[, ,] for three-dimensional arrays

8 Interrupt handling

When using robots in complex manufacturing systems, it is necessary for the robot to be able to react specifically and immediately to certain external or internal events and for the execution of other actions parallel to the robot process to be possible. In other words, a running robot program must be interrupted and an interrupt subprogram or function started. After the subprogram has been executed, and if nothing further is declared, the interrupted robot program should be resumed.

This specific interruption or starting of a program is made possible by the interrupt statement. In this way the user has the possibility of reacting by program to an event which does not occur synchronously with execution of the program.

Interrupts can be triggered by

- equipment such as sensors, peripheral units, etc.,
- error messages
- the user, or
- safety circuits.

For example, an interrupt routine which resets certain output signals (prepared program `IR_STOPM.SRC`) can be called when an Emergency Stop button is pressed.

8.1 Declaration

The possible causes of interruption, and the respective ways the system should react to them, must be defined before an interrupt can be activated.

INTERRUPT This is done using the interrupt declaration, in which every interrupt must be assigned a priority, an event and the interrupt routine to be called. The complete syntax is:

```
INTERRUPT DECL Priority WHEN Event DO Subprogram
```

For the meaning of the arguments, see Table 25.

Argument	Data type	Meaning
Priority	INT	Arithmetic expression specifying the priority of the interrupt. Priority levels 1...39 and 81...128 are available. The values 40...80 are reserved for automatic priority allocation by the robot system. A level 1 interrupt has the highest priority.
Event	BOOL	Logical expression defining the interrupt event. The following are permissible: <ul style="list-style-type: none"> • a boolean constant • a boolean variable • a signal name • a comparison
Subprogram		The name of the interrupt program to be executed when the event occurs.

Table 25 Arguments in the interrupt declaration

The instruction

```
INTERRUPT DECL 4 WHEN $IN[3]==TRUE DO UP1()
```

declares a priority 4 interrupt, for example, which is called by the subprogram `UP1()` as soon as input 3 is set to High.

The interrupt declaration is an instruction. It must not, therefore, be located in the declaration section!

An interrupt is only recognized at, or below, the programming level in which it is declared. At higher programming levels, despite being activated, the interrupt is not recognized. In other words, an interrupt declared in a subprogram is not recognized in the main program (see Fig. 38).

GLOBAL

If, on the other hand, the interrupt is declared as **GLOBAL**, it can be declared in any subprogram and does not lose its validity when this level is left (see Fig. 38).

```
GLOBAL INTERRUPT DECL 4 WHEN $IN[3]==TRUE DO UP1()
```



- A declaration may be overwritten by another at any time.
- A GLOBAL interrupt differs from a normal interrupt in that it remains valid even after the subprogram in which it was declared has been left.
- Up to 32 interrupts may be declared at any one time.
- Structure variables and components may not be accessed in the interrupt condition.
- Runtime variables may not be transferred as interrupt routine parameters, apart from GLOBAL variables or variables declared in the data list.

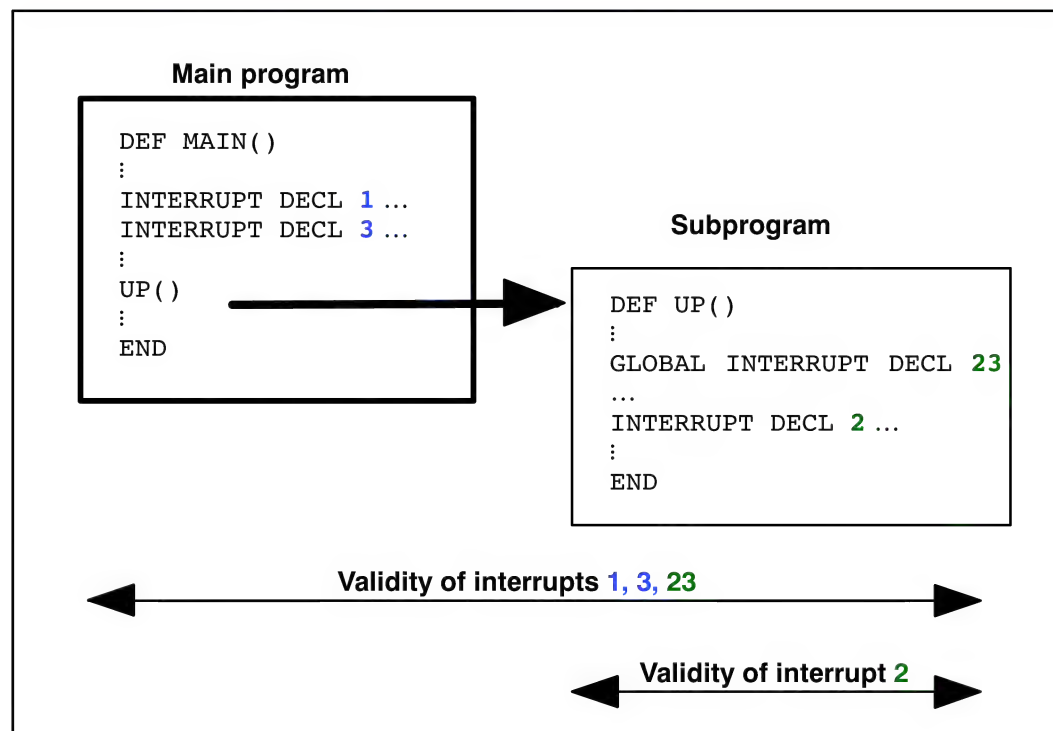


Fig. 38 Validity range for an interrupt dependent on place and type of declaration

8.2 Activating interrupts

Switch interrupt on	<p>When first declared, an interrupt is switched off. Using the instruction</p> <pre>INTERRUPT ON 4</pre> <p>the interrupt with priority 4 is switched on, whereas</p> <pre>INTERRUPT ON</pre> <p>switches all interrupts on. No reaction to the defined interrupt can take place until the interrupt has been switched on. The interrupt event is now cyclically monitored.</p>
Edge-triggered	<p>This checking is edge-triggered, i.e. an interrupt is only triggered if the logic condition changes from the FALSE state to the TRUE state, and not, however, if the condition is already TRUE when the interrupt is switched on.</p>
<p>For reasons of runtime, only 16 interrupts can be switched on at any one time. Bear this particularly in mind in the case of global activation of all interrupts.</p>	
Switch interrupt off	<p>In the same way as they are switched on, interrupts can also be switched off either individually or all together:</p> <pre>INTERRUPT OFF 4</pre> <p>or</p> <pre>INTERRUPT OFF</pre>
Disable / enable	<p>Using the keywords <code>ENABLE</code> and <code>DISABLE</code>, interrupts that have been switched on can be enabled or disabled, individually or globally.</p> <p>The disable command makes it possible to protect certain sections of the program before it is interrupted. A disabled interrupt will be recognized and saved but not executed. The interrupts that have occurred are executed in order of their priority, but only once they have been enabled.</p> <pre>DISABLE 4</pre> <p>or</p> <pre>DISABLE</pre> <p>There is no further reaction to an event that has been saved if the interrupt is switched off before triggering. If an interrupt occurs several times while it is disabled, it is only executed once on being enabled.</p>
<p>The preconditions for triggering an interrupt are:</p> <ul style="list-style-type: none"> • The interrupt must be declared (<code>INTERRUPT DECL ...</code>) • The interrupt must be switched on (<code>INTERRUPT ON</code>) • The interrupt must not be disabled • The corresponding event must have occurred (edge-triggered) 	
Priority	<p>If several interrupts occur at the same time, the interrupt with the highest priority is processed first, then those of lower priority. Priority level 1 here has the highest priority and level 128 the lowest.</p> <p>When an event is recognized, the current actual position of the robot is saved and the interrupt routine called. The interrupt that has occurred is disabled throughout the time it is being executed along with all those of lower priority. When returning from the interrupt program, this so-called implicit disabling is cancelled, including that of the current interrupt.</p> <p>The interrupt can now be executed again if the event recurs (even during the interrupt program). If this is prevented, the interrupt must be explicitly disabled or switched off before returning to the main program.</p>

At any point after the first command in the interrupt program, an interrupt can itself be interrupted by an interrupt with a higher priority. In the first command, the programmer has the possibility of preventing this by disabling or switching off one or all of the interrupts. If an interrupt switches itself off in the interrupt program, the interrupt program is, of course, executed through to the end.

When a higher-priority interrupt is terminated, the interrupted interrupt program is resumed at the point at which it was interrupted.



- IN parameters can be transferred to an interrupt program.
- If a local interrupt program is to send back a parameter, the variable must be declared in the data list of the main program. In the case of global interrupt programs, the data list `$CONFIG.DAT` must be used.
- Changes to `$TOOL` and `$BASE` in the interrupt program are only effective there (command mode).
- There is no computer advance run in the interrupt program because it runs at command level, i.e. it is executed block by block (\Rightarrow `$ADVANCE` assignments are not permissible). Approximate positioning is thus not possible.

Special cases:

- Interrupts to the system variables `$ALARM_STOP` and `$STOPMESS` are also executed in the event of an error, i.e. the interrupt statements are executed despite the robot being stopped (motion instructions are disregarded).
- Any interrupt that has been declared and activated can be recognized during an operator stop. Once the program is restarted, interrupts that have occurred are executed in order of priority (if enabled) and the program then continues.

A robot motion that is already being executed is not interrupted by an interrupt program call. Even while the interrupt program is being processed, all motions already prepared in the interrupted program are still executed. If the interrupt program is completely executed during this time, the program that was interrupted is resumed without any pause between motions, i.e. without the processing time being lengthened. If, on the other hand, the interrupt action is not yet complete, the robot remains motionless until the next motion is prepared and continued after returning to the main program.

If the interrupt program itself contains motion instructions, it stops at the first motion instruction until the advance run in the main program has been executed.

The following example is designed to explain the use of interrupt statements and the use of specific system variables. Two sensors (at inputs 1 and 2) are constantly monitored here during a linear motion. As soon as a sensor detects a part (is set to High), an interrupt subprogram is called which saves the position of the part and sets the corresponding output as an indicator. The robot motion is not interrupted here. The robot then moves again to the parts that have been detected.



```

DEF  INTERRUPT ( )

;----- Declaration section -----
EXT  BAS (BAS_COMMAND :IN,REAL :IN )
DECL AXIS HOME
INT I

;----- Initialization -----
BAS (#INITMOV,0 ) ;Initialization of velocities,
                    ;accelerations, $BASE, $TOOL, etc.
HOME={AXIS: A1 0,A2 -90,A3 90,A4 0,A5 30,A6 0}
FOR I=1 TO 16
    $OUT[I]=FALSE ;Reset all outputs
ENDFOR
INTERRUPT DECL 10 WHEN $IN[1]==TRUE DO SAVEPOS (1 )
INTERRUPT DECL 11 WHEN $IN[2]==TRUE DO SAVEPOS (2 )

;----- Main section -----
PTP  HOME ;BCO run

PTP  {X 1320,Y 100,Z 1000,A -13,B 78,C -102}

INTERRUPT ON ;Activate all interrupts
LIN  {X 1320,Y 662,Z 1000,A -13,B 78,C -102} ;Search path
INTERRUPT OFF 10 ;Switch off interrupt 10
INTERRUPT OFF 11 ;Switch off interrupt 11

PTP  HOME

FOR I=1 TO 2
    IF $OUT[I] THEN
        LIN  PART[I] ; Move to detected part
        $OUT[I]=FALSE
        PTP  HOME
    ENDIF
ENDFOR

END

;----- Interrupt program -----
DEF  SAVEPOS (NR :IN ) ;Part detected
INT NO
$OUT[NO]=TRUE          ;Set flag
PART[NO]=$POS_INT      ;Save position
END

```



Along with the basic package (BAS.SRC), another file, IR_STOPM(), comes as standard in the controller. This subprogram executes a number of fundamental instructions in the event of an error. These include several technology-specific procedures and the repositioning of the robot on the motion path. This is because whereas the robot stays on the path when the EMERGENCY STOP button is pressed, hardware-triggered stops for safeguards directly affecting the operator (e.g. safety gate) are not true to the path. You should, therefore, always implement the following sequence in the initialization section of your program (located as standard in the BAS INI fold):

```

INTERRUPT DECL 3 WHEN $STOPMESS==TRUE DO IR_STOPM ( )
INTERRUPT ON 3

```

The PTP \$POS_RET instruction in the file IR_STOPM() brings about the repositioning and thus re-establishes BCO.

Further system variables useful for working with interrupts are shown in Table 26. The positions are always related to the current coordinate systems in the main run.

Joint (axis-specific)	Cartesian	Description
\$AXIS_INT	\$POS_INT	Position at which the interrupt was triggered
\$AXIS_ACT	\$POS_ACT	Current actual position
\$AXIS_RET	\$POS_RET	Position at which the robot left the path
\$AXIS_BACK	\$POS_BACK	Position of the start point of the path
\$AXIS_FOR	\$POS_FOR	Position of the destination point of the path

Table 26 Useful system variables for interrupt handling

In motions with approximate positioning, the positions ..._BACK and ..._FOR are dependent on the location of the main run. For more information see Fig. 39 and Fig. 40.

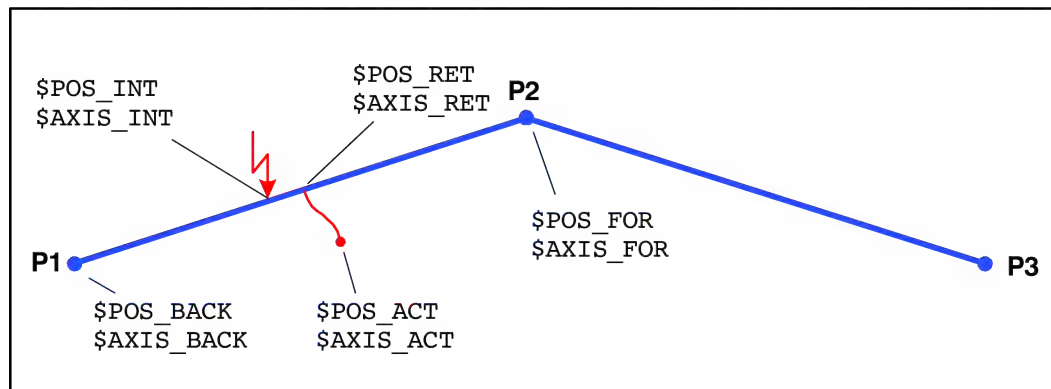


Fig. 39 Interrupt system variables with exact positioning points

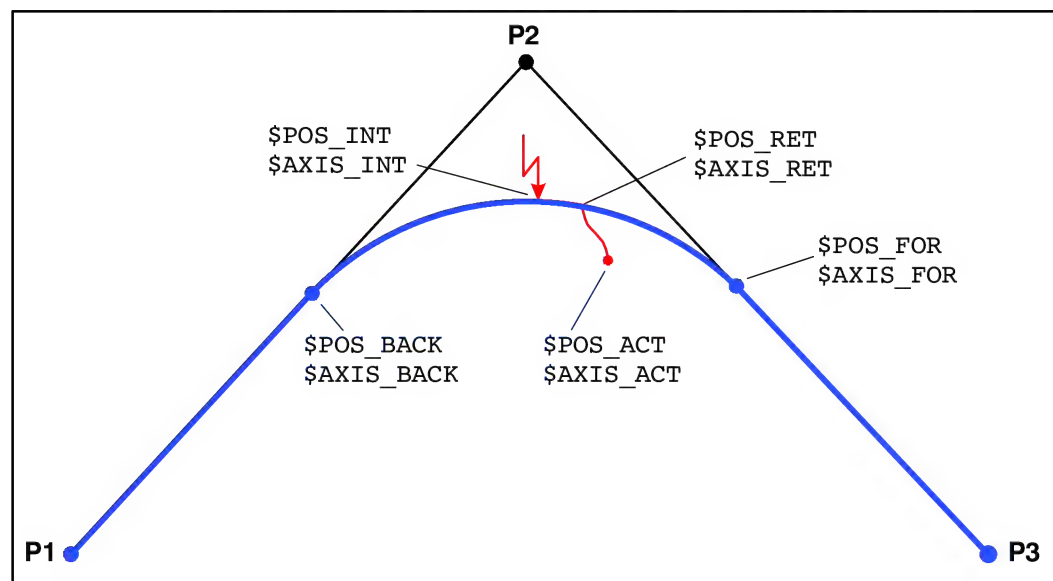


Fig. 40 Interrupt system variables in the event of interrupt in an approximate positioning range

8.3 Stopping active motions

BRAKE

If it is necessary, in the event of an interrupt, to be able to stop robot motions that are already being executed, the **BRAKE** statement is used in the interrupt program. Programming without

BRAKE

parameters causes the motion to be braked using the programmed path or axis acceleration values. The reaction is the same as that when the **STOP** key is pressed. The robot does not leave the motion path.

Using the instruction

BRAKE F

(brake fast) a shorter braking distance is achieved. The robot stops in the shortest time possible while remaining true to the path.

The **BRAKE** instruction must only be placed in an interrupt program. In other programs it leads to an error-induced stop.

The **BRAKE** instruction does not have to come straight after the call, but can be situated at any position in the interrupt program. The effect it has depends on whether or not a motion is still being carried out in the interrupted program when it is executed. If the robot is stationary, the instruction has no effect. A motion still in progress in the interrupted program is stopped using the programmed braking mode. The **BRAKE** command is no substitute for the **HALT** statement, however, if program execution is to be stopped. Processing of the interrupt program only continues with the following instruction once the robot has come to a standstill.



After returning to the interrupted program, a motion stopped by means of **BRAKE or **BRAKE F** in the interrupt program is resumed!**

8.4 Cancelling interrupt routines

During the robot motion in example 8.1, proximity switches detect up to 2 objects and record their positions so the robot can move to them again later.

Even if both objects are already known, the robot nonetheless moves along the complete search path. In order to save time, it is desirable to terminate the movement as soon as the maximum number of parts have been detected.

RESUME

Cancellation of a robot motion is possible with the KR C... using the
RESUME

statement. RESUME cancels all running interrupt programs and subprograms up to the level at which the current interrupt was declared.

Just like the BRAKE statement, RESUME is only permissible in an interrupt program.

When the RESUME statement is activated, the advance run pointer must not be at the level where the interrupt was declared, but at least one level lower.

Since RESUME is intended to cancel the search path, the search motion must be programmed in a subprogram. In the following example, this is accomplished in MOVEP (); the interrupt subprogram is called IR_PROG ().

It is important for the advance run to be stopped before the last line in subprograms that are to be cancelled using RESUME. Only then is it possible to ensure that, when the RESUME statement is activated, the advance run pointer will not be at the level where the interrupt was declared. In MOVEP (), this was done using the \$ADVANCE=0 assignment.

In the interrupt program itself, the search path is stopped using the BRAKE statement, as soon as a sensor at input 1 has detected 4 parts, and then cancelled by means of the RESUME statement (since MOVEP () is also terminated along with IR_PROG ()). Without the BRAKE statement, the search motion would still be executed in the advance run.

After the RESUME statement, the main program is resumed at the instruction following the subprogram call, i.e. \$ADVANCE=3 (reset advance run).



```

DEF SEARCH ( )
;----- Declaration section -----
EXT BAS (BAS_COMMAND :IN,REAL :IN )
DECL AXIS HOME
;----- Initialization -----
INTERRUPT DECL 3 WHEN $STOPMESS==TRUE DO IR_STOPM ( )
INTERRUPT ON 3 ;standard fault service functions
BAS (#INITMOV,0 ) ;Initialization of velocities,
;accelerations, $BASE, $TOOL, etc.
HOME={AXIS: A1 0,A2 -90,A3 90,A4 0,A5 30,A6 0}
INTERRUPT DECL 11 WHEN $IN[1] DO IR_PROG ( )
I[1]=0 ;Set predefined counter to 0
;----- Main section -----
PTP HOME ;BCO run
INTERRUPT ON 11
MOVEP ( ) ;Motion along the search path
$ADVANCE=3 ;Reset advance run
INTERRUPT OFF 11
GRIP ( )
PTP HOME
END
;----- Subprogram -----
DEF MOVEP ( ) ;Subprogram for search path motion
PTP {X 1232,Y -263,Z 1000,A 0,B 67,C -90}
LIN {X 1232,Y 608,Z 1000,A 0,B 67,C -90}
$ADVANCE=0 ;Stop advance run
END ;
;----- Interrupt program -----
DEF IR_PROG ( ) ;Save position of parts
;INTERRUPT OFF 11
I[1]=I[1]+1
POSITION[I[1]]=$POS_INT ;Position saved
IF I[1]==4 THEN ;4 parts recognized
BRAKE ;Motion stopped
RESUME ;IR_PROG & MOVE cancelled
ENDIF
;INTERRUPT ON 11
END
;----- Subprogram -----|

DEF GRIP ( ) ;Grasp detected parts
INT POS_NO ;Counter variable
FOR POS_NR=I[1] TO 1 STEP -1
POSITION[POS_NR].Z=POSITION[POS_NR].Z+200
LIN POSITION[POS_NR] ;Move to 200 mm above part
LIN REL {Z -200} ;Move vertically to part
; Pick part up
LIN POSITION[POS_NR] ;Move back up
LIN {X 634,Y 1085,Z 1147,A 49,B 67,C -90}
; Set part down
ENDFOR
END

```



If there is a risk of an interrupt being incorrectly triggered twice because of sensitive sensors (“contact bouncing”), you can prevent this by switching off the interrupt in the first line of the interrupt program. However, a genuine interrupt arising during interrupt processing can now no longer be recognized. If the interrupt is to remain active, it must be switched back on before returning to the main program.



If a motion is cancelled using the `RESUME` statement, as in the example above, the following motion should not be a `CIRC` motion because the start point will be different every time (\Rightarrow different circles).

In the search action programmed in example 8.2, the inputs are polled at the interpolation cycle rate (currently 12 ms). The maximum degree of inaccuracy is thus around 12 ms times the path velocity.

“Fast measurement”

If you wish to avoid this degree of inaccuracy, you must not connect the proximity switch to the user inputs; instead, you must use the 4 special inputs on the peripheral connector X11. These inputs can be addressed via the system variables `$MEAS_PULSE[1]...` `MEAS_PULSE[4]` (reaction time 125 μ s).

The measurement pulse must not currently be applied when the interrupt is switched on, otherwise the corresponding error message appears.

8.5 Use of cyclical flags

No logical operations are permissible in the interrupt declaration instruction.

In order, therefore, to be able to define complex events, you must work with cyclical flags, as only these make constant updating possible.

With the program sequence

```
:  
$CYCFLAG[3] = $IN[1] AND ([ $IN[2] OR $IN[3] )  
INTERRUPT DECL 10 WHEN $CYCFLAG[3] DO IR_PROG()  
INTERRUPT ON 10  
:
```

you can simultaneously monitor and logically combine 3 inputs.



Further information can be found in the chapter **[Variables and declarations]**, section **[System variables and system files]**.

9 Trigger – Path-related switching actions

Unlike the interrupt functions, which are independent of robot motion, some applications also require switching actions which are triggered depending on the motion path. Such applications include, e.g.:

- Closing or opening the welding gun during spot welding
- Switching the welding current on/off during arc welding
- Starting or stopping the flow of adhesive in bonding or sealing applications.

In the KR C..., these path-related switching actions are possible using the **TRIGGER** statement. Parallel to the next robot motion, it is possible, by means of **TRIGGER**, and in accordance with a path criterion, to execute a subprogram, assign a value to a variable or **PULSE** statement, or set an output.

9.1 Switching action at the start or end point of the path

TRIGGER

If a switching action relating to the start or end point of a motion path is required, program a **TRIGGER** statement before the relevant motion instruction (**PTP**, **LIN** or **CIRC**) using the following syntax:

```
TRIGGER WHEN DISTANCE=switching point DELAY=time DO  
instruction PRIO=priority
```

The arguments are described in greater detail in the following table.

Argument	Data type	Meaning
Switching point	INT	In the case of individual blocks , DISTANCE=0 designates the start point and DISTANCE=1 the end point of the following motion. In approximation blocks , the specification DISTANCE=1 signifies the middle of the subsequent approximate positioning arc. If the previous block is already an approximation block, DISTANCE=0 signifies the end point of the preceding approximate positioning arc.
Time	INT	Using the specification DELAY , it is possible to delay or advance the switching point by a certain amount of time. The switching point can, however, only be delayed or advanced in so far as it still remains in the block concerned. The unit is milliseconds .
Instruction		The instruction can be <ul style="list-style-type: none"> • a subprogram call • the assignment of a value to a variable • an OUTPUT instruction (also Pulse).
Priority	INT	Every TRIGGER statement with a subprogram call must be assigned a priority. Values from 1...39 and 81...128 are permissible. The priorities are thus the same as those for interrupts (see chapter 8). The values 40...80 are reserved for automatic priority allocation by the robot system. For this, program PRIO=-1 .

Table 27 Arguments in the **TRIGGER** statement

Using the instruction sequence

```

:
LIN POINT2
:
TRIGGER WHEN DISTANCE = 0 DELAY=20 DO $OUT[4]=TRUE
TRIGGER WHEN DISTANCE = 1 DELAY=-25 DO UP1() PRIO=-1
LIN POINT3
:
LIN POINT4
:

```

output 4 is set exactly 20 milliseconds after the start of the linear motion to POINT3, and the subprogram UP1() is called 25 milliseconds before the end point is reached. Priorities are automatically allocated by the system.

For an explanation of the different effects of the specification DISTANCE on individual blocks and approximate positioning blocks see Fig. 41 – Fig. 44.

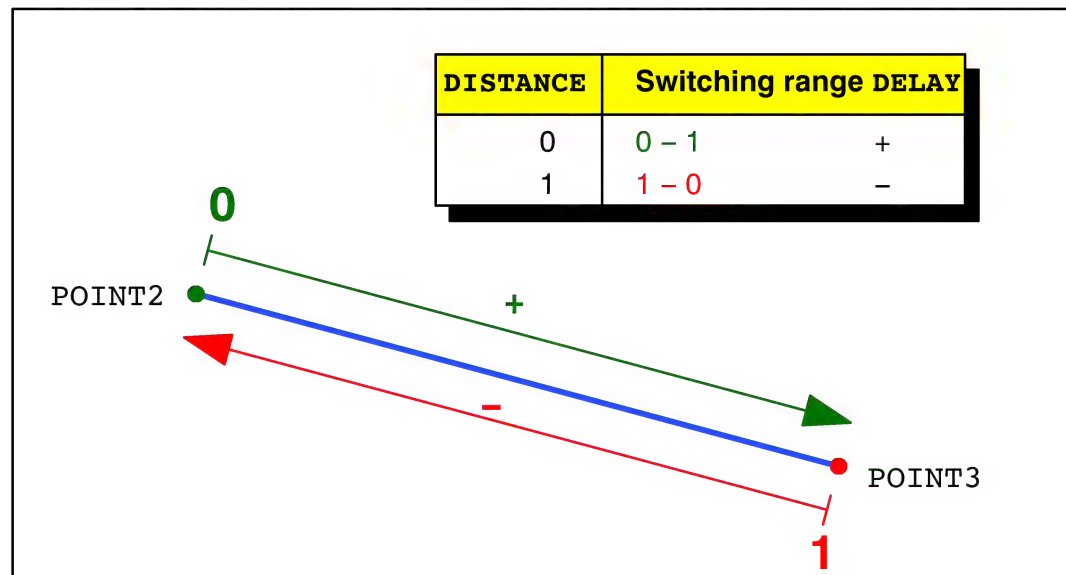


Fig. 41 Switching ranges and possible delay values if start and end points are exact positioning points

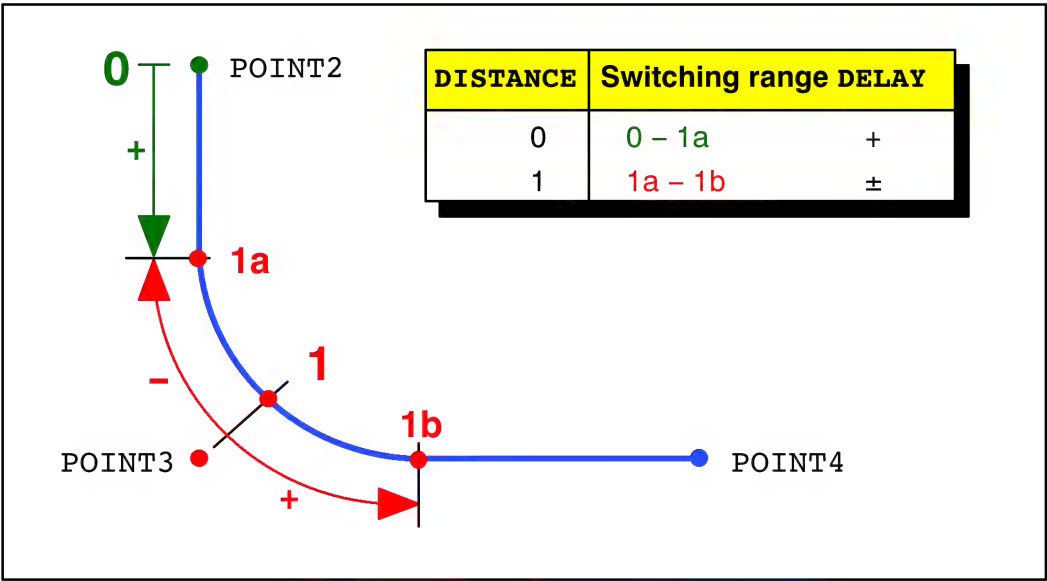


Fig. 42 Switching ranges and possible delay values if the start point is an exact positioning point and the end point is an approximate positioning point

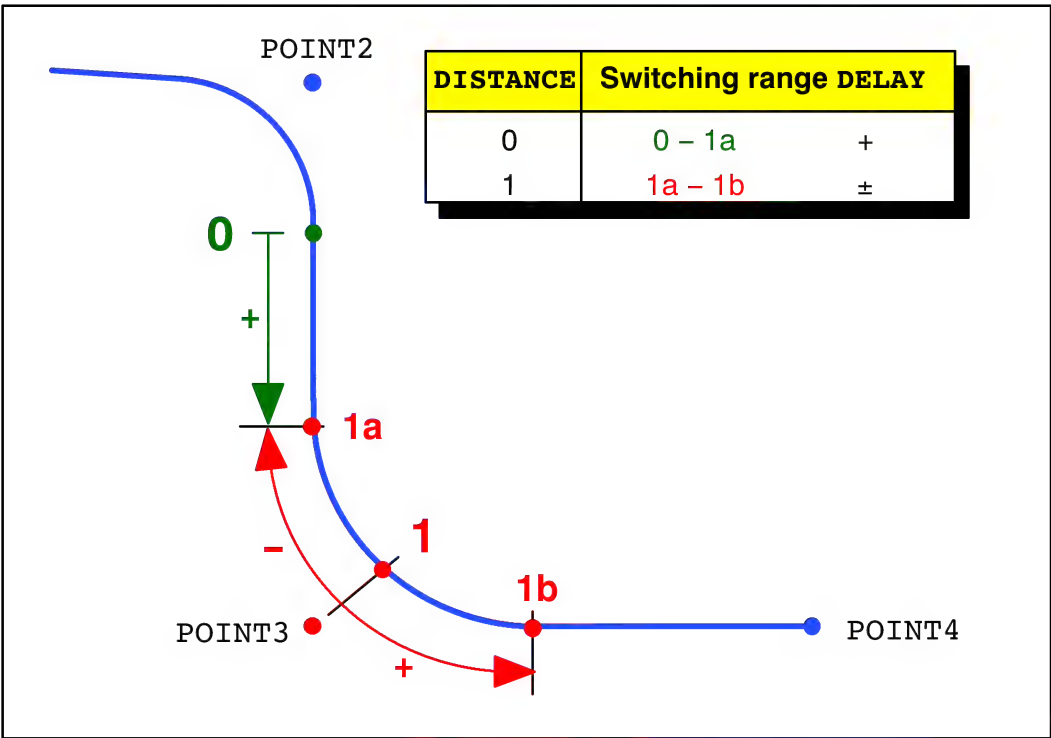


Fig. 43 Switching ranges and possible delay values if start and end points are approximate positioning points

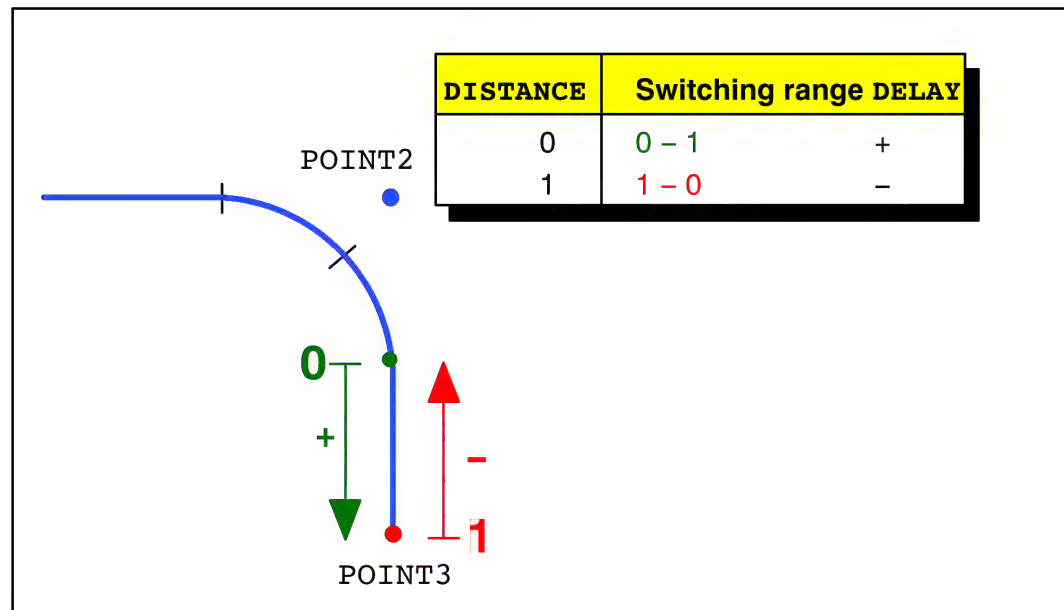


Fig. 44 Switching ranges and possible delay values if the start point is an approximate positioning point and the end point is an exact positioning point

9.2 Switching action at any point on the path

If you are using the path-related TRIGGER statement, you can trigger the switching action at any position along the path by specifying a distance. As with switching actions at the start or end points, this again can additionally be delayed or brought forward.

The path-related switching action is only allowed with continuous-path motions (LIN or CIRC). The TRIGGER statement refers here to the next programmed motion block and has the following syntax:

```
TRIGGER WHEN PATH = distance DELAY = time DO instruction
<PRIO=priority
```

The arguments are described in greater detail in the following table.

Argument	Data type	Meaning
Distance	INT	<p>With distance you can specify the desired distance from the end point programmed after the trigger.</p> <p>If this end point is an approximated point, distance specifies the desired distance of the switching action from the position in the approximate positioning range closest to the end point.</p> <p>The switching point can be shifted back as far as the start point by entering a negative distance. If the start point is an approximate positioning point, the switching point can be shifted as far as the start of the approximate positioning range.</p> <p>By entering a positive distance, a shift as far as the next exact positioning point programmed after the trigger is possible.</p> <p>The unit is millimeters.</p>
Time	INT	<p>Using the specification DELAY, it is possible to delay (+) or advance (–) the switching point relative to the PATH specification by a certain amount of time.</p> <p>The switching point can only be delayed or advanced, however, in the switching range given above (as far as the next exact positioning point). With approximate positioning motions, the switching point can be advanced, at most, as far as the start of approximate positioning of the start point.</p> <p>The unit is milliseconds.</p>
Instruction		<p>The instruction can be</p> <ul style="list-style-type: none"> • a subprogram call • the assignment of a value to a variable • an OUTPUT instruction (also Pulse).
Priority	INT	<p>Every TRIGGER statement with a subprogram call must be assigned a priority. Values from 1...39 and 81...128 are permissible. The priorities are thus the same as those for interrupts (see chapter 8).</p> <p>The values 40...80 are reserved for automatic priority allocation by the robot system. For this, program PRIO=–1.</p>

Table 28 Arguments in the TRIGGER statement

Instruction sequence:

```

:
LIN POINT2 C_DIS
  TRIGGER WHEN PATH = Y DELAY= X DO $OUT[2]=TRUE
LIN POINT3 C_DIS
LIN POINT4 C_DIS
LIN POINT5
:

```

Since the switching point can be shifted from the motion point before which it was programmed, past all subsequent approximate positioning points, as far as the next exact positioning point, it is possible to shift the switching point from the approximate positioning start point POINT2 to POINT5. If POINT2 was not approximated in this sequence of instructions, the switching point could only be shifted as far as the exact positioning point POINT2.

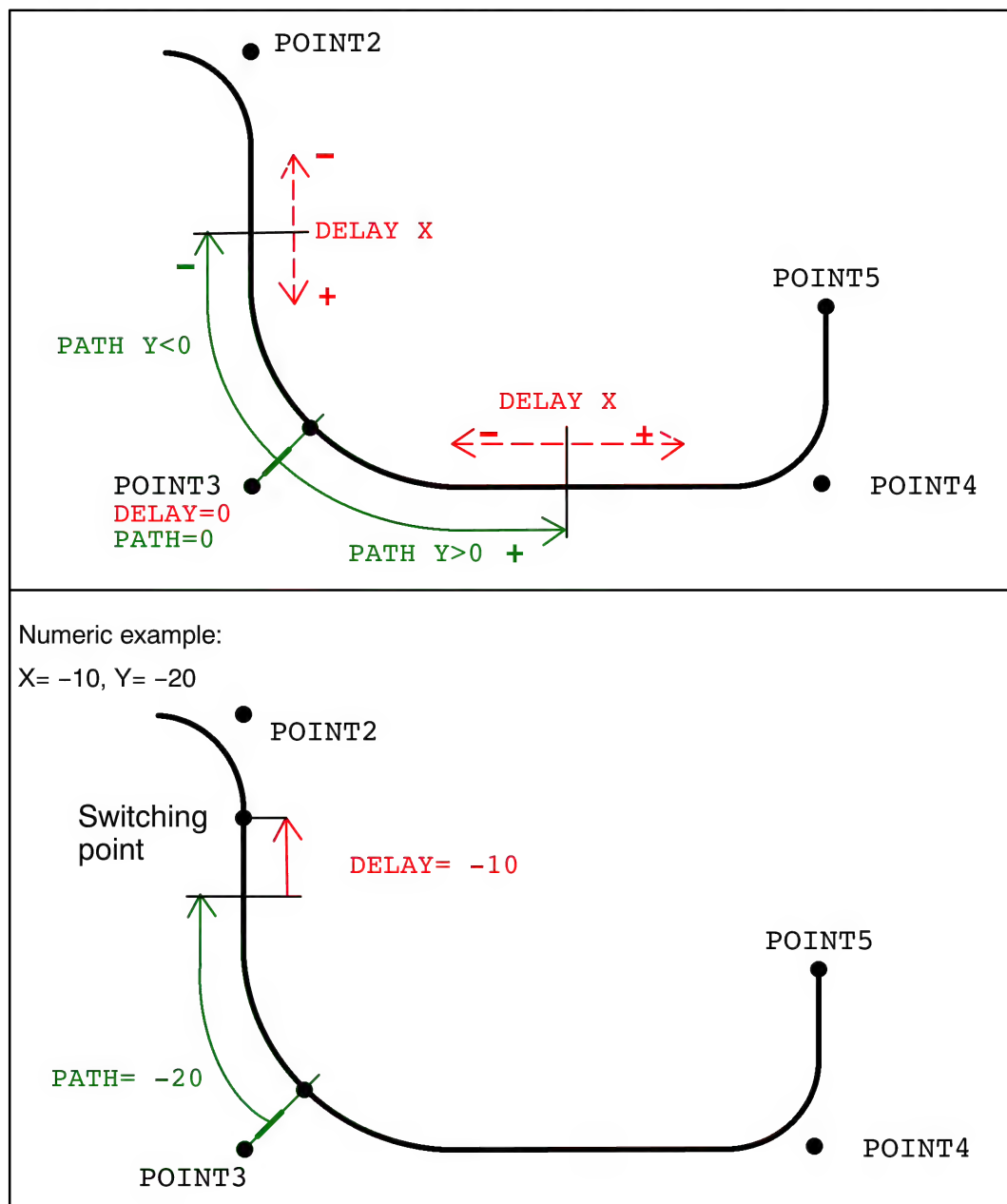


Fig. 45 Switching ranges if the start point is an approximate positioning point

**Special cases:**

- **BCO run**

If block selection is carried out to a continuous-path motion, this movement is performed as a BCO run. Since the start point for this BCO run is entirely arbitrary, it can be of no use as the start point for a distance specification. Therefore, if TRIGGER commands with PATH specification are programmed before such a motion, and if a block selection is made to these commands, they will all be executed at the end point.

- **Approximation not possible**

If approximation is not possible; an exact positioning motion is carried out at this position. In this context, however, it is treated in the same way as an approximate positioning motion. Switching actions further along the path remain saved and are triggered at the relevant position. Generally speaking, however, they will no longer be exactly as programmed, since the path, and thus the path length, are now different. Switching actions set in the first half of the approximate positioning range by means of a negative PATH value cannot now be triggered before the approximate positioning point:

```

:
LIN P1 C_DIS
TRIGGER WHEN PATH=-120 DELAY=0 DO UP1() PRIO=-1
TRIGGER WHEN PATH=-70 DELAY=0 DO $OUT[2]=TRUE
LIN P2 C_DIS
:

```

In the above example, the distance between the start and end points should be 100 mm. If approximation is possible for P1, the subprogram call UP1 () is executed 20 mm before the point on the path nearest to the approximate positioning point P1 is reached. Output 2 is set 30 mm after this point on the path. If it was not possible to carry out approximation for P1, the path runs through P1, where exact positioning takes place. The subprogram call UP1 () is now executed immediately after leaving P1 and output 2 is set at a distance of 30 mm from P1.

- **Cancelling a motion**

If a motion is cancelled, by block selection or reset, for example, and not subsequently completed, switching actions that have not yet been executed will be deleted, not executed, in the event of a DISTANCE specification.

- **Path-related TRIGGER statement for a PTP motion**

If a PATH-TRIGGER statement with path specification is programmed for a PTP motion, this will be refused by the interpreter when the motion is executed.

- **PTP-CP approximate positioning**

If a PATH-TRIGGER statement is programmed for a motion whose start point is a PTP-CP approximate positioning point, the switching action can take place, at the earliest, at the end of this approximate positioning range, since the whole approximate positioning range is now covered using a PTP motion.

In the event of a CP-PTP approximate positioning range, all TRIGGER statements which are still active, but have not yet been switched on, are triggered at the start point of the approximate positioning range. This is because the motion is continued as PTP from this point and path assignment is no longer possible.

In the next example, switching actions with **DISTANCE** specifications and also with **PATH** specifications are programmed. The individual switching points and the motion path are illustrated in Fig. 46.



```

DEF TRIG ( )
;----- Declaration section -----
EXT BAS (BAS_COMMAND :IN,REAL :IN)
DECL AXIS HOME
INT I
SIGNAL GLUE $OUT[3]
;----- Initialization -----
INTERRUPT DECL 3 WHEN $STOPMESS==TRUE DO IR_STOPM ( )
INTERRUPT ON 3
BAS (#INITMOV,0 ) ;Initialization of velocities,
                  ;accelerations, $BASE, $TOOL, etc.
$APO.CDIS=35      ;Define approximation distance
HOME={AXIS: A1 0,A2 -90,A3 90,A4 0,A5 30,A6 0}
POS0={POS: X 1564,Y -114,Z 713,A 128,B 85,C 22,S 6,T 50}
POS1={X 1383,Y -14,Z 713,A 128,B 85,C 22}
POS2={X 1383,Y 200,Z 713,A 128,B 85,C 22}
POS3={X 1527,Y 200,Z 713,A 128,B 85,C 22}
POS4={X 1527,Y 352,Z 713,A 128,B 85,C 22}
FOR I=1 TO 16
    $OUT[I]=FALSE
ENDFOR
;----- Main section -----
PTP HOME ;BCO run
PTP POS0
TRIGGER WHEN DISTANCE=0 DELAY=40 DO $OUT[1]=TRUE
TRIGGER WHEN PATH=-30 DELAY=0 DO UP1(2) PRIO=-1
LIN POS1
TRIGGER WHEN DISTANCE=1 DELAY=-50 DO GLUE=TRUE
TRIGGER WHEN PATH=180 DELAY=55 DO PULSE($OUT[4],TRUE,0.9)
TRIGGER WHEN PATH=0 DELAY=40 DO $OUT[6]=TRUE
LIN POS2 C_DIS
TRIGGER WHEN DISTANCE=0 DELAY=40 DO PULSE ($OUT[5],TRUE,1.4 )
TRIGGER WHEN PATH=-20 DELAY=-15 DO $OUT[8]
LIN POS3 C_DIS
TRIGGER WHEN DISTANCE=1 DELAY=-15 DO UP1 (7 ) PRIO= -1
LIN POS4
PTP HOME
END
DEF UP1 (NR :IN )

INT NR
IF $IN[1]==TRUE THEN
    $OUT[NR]=TRUE
ENDIF
END

```

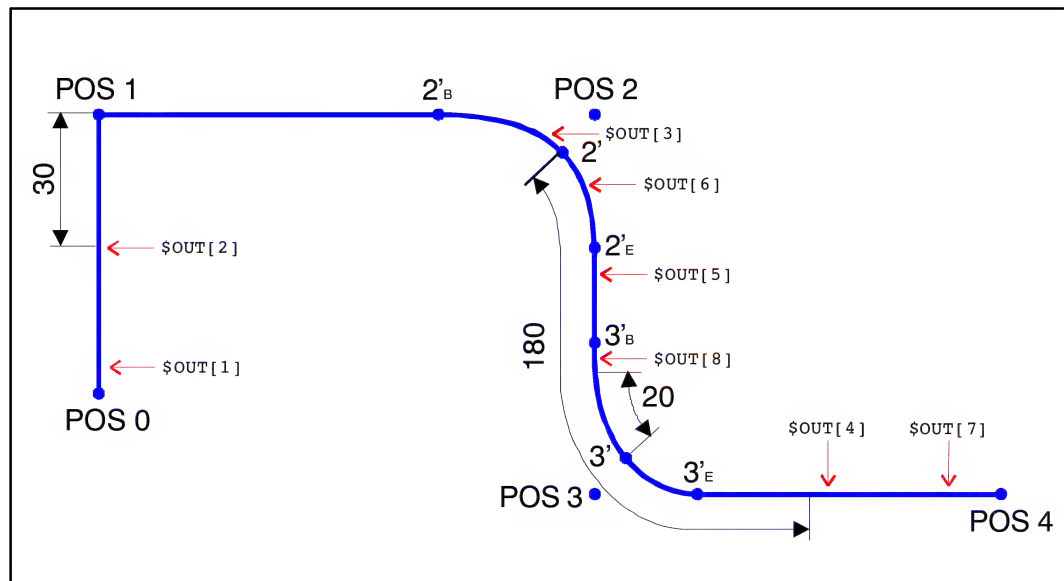


Fig. 46 Switching points and motion path for the above example

9.3 Tips and tricks

9.3.1 Overlapping Trigger statements

Although it is possible to run the program illustrated, problems may arise due to the identical priority of the Trigger statements.



```
;FOLD PTP P1
TRIGGER WHEN DISTANCE=0 DELAY=0 DO SUB1() PRIO=15
PTP P1
...
;ENDFOLD
...
;FOLD PTP P2
TRIGGER WHEN DISTANCE=0 DELAY=-75 DO SUB1() PRIO=15
...
;ENDFOLD
```

If the first Trigger statement has not been terminated before the second Trigger statement is activated, a runtime error message is generated. This can happen, for example, if the two points are close together.



There are two possible ways of solving this problem:

- Assign each of the two Trigger statements a priority manually;
- Assign both of the Trigger statements the priority “-1”, so that the system then automatically assigns the correct priority.

10 Data lists

10.1 Local data lists

Data lists are used for preparing program-specific and higher-level declarations. These include information about points, e.g. coordinates:

- One data list may be drawn up for each SRC file. This has the same name as the SRC file and ends with the extension “.DAT”.
- The data list is local even though it is a separate file in its own right.
- A data list may **only** consist of declarations and initializations.
- A single line can consist of declarations and initializations.
- System variables are not accepted.

DEFDAT

The declaration of data lists is analogous to that of SRC files: The declaration is introduced using the keyword `DEFDAT` and the program name and concluded with the keyword `ENDDAT`.

Variables are initialized by assigning a value directly to the variable concerned in the declaration line.

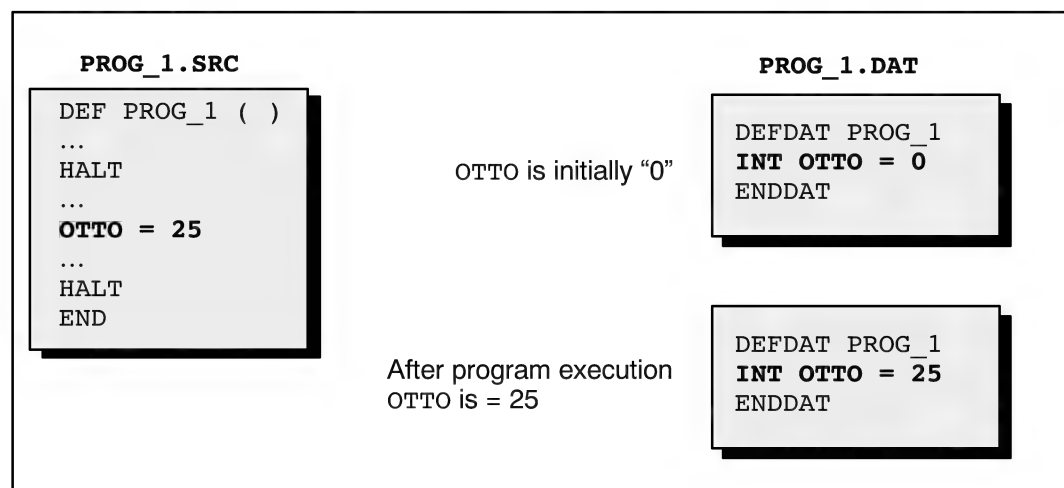


Fig. 47 Initialization and assignment of values to variables declared in data lists

Declaration and initialization in the data list eliminates the need for this in the main program. If a new value is assigned to the variable `OTTO` in the main program, it is also entered into the data list and permanently saved (see Fig. 47).

The “new” value is thus used after the controller has been switched off and back on again. This is essential for online correction and other program modifications.

If a main program always starts with the same value, the relevant variable in the main program must be preset with the desired value.

The following declarations are allowed in data lists:

- External declarations for subprograms and functions which are used in the SRC file.
- Import declarations for imported variables.
- Declarations and initializations of variables which are used in the SRC file.
- Declarations of signal and channel names which are used in the SRC file.
- Declarations of data and enumeration types (Struc, Enum) which are used in the data list or in the SRC file.

10.2 Global data lists

Variables defined in a data list can be made accessible to a “foreign” main program.

PUBLIC

To do this, the data list must be defined as “publicly accessible” with the keyword **PUBLIC** in the header line. There are now two possible ways of declaring variables:

IMPORT

- A variable is defined in the data list, e.g. as `INT OTTO = 0`, and must be imported into the “foreign” main program using the command `Import` in order to be accessible.

An imported variable can be given a different name in the main program from the one it had in the data list from which it was imported.

If you want to use the variable `OTTO`, taken from the above data list `PROG_1`, in program `PROG_2 ()`, you thus program the following import declaration in `PROG_2 ()`, as well as the keyword **PUBLIC** in the data list:

```
IMPORT INT OTTO_2 IS /R1/PROG_1..OTTO
```

The variable `OTTO` from the data list `PROG_1.DAT` in the directory `/R1` is now also known as `OTTO_2` in the program `PROG_2 ()` (see Fig. 48).

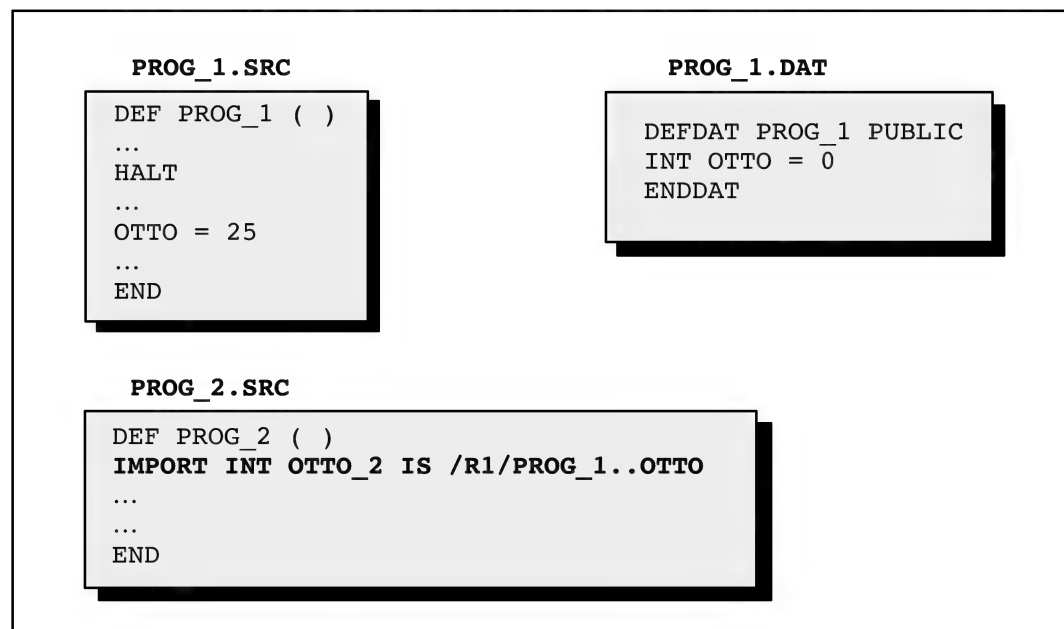


Fig. 48 Importing variables from “foreign” data lists with `Import`

GLOBAL

- The variable is declared as a “global variable”, e.g. `DECL GLOBAL INT OTTO = 0`, and is accessible to all foreign main programs without the need for the `Import` command.

If a global variable has been declared, it is not possible to change the name of the variable in a foreign main program.

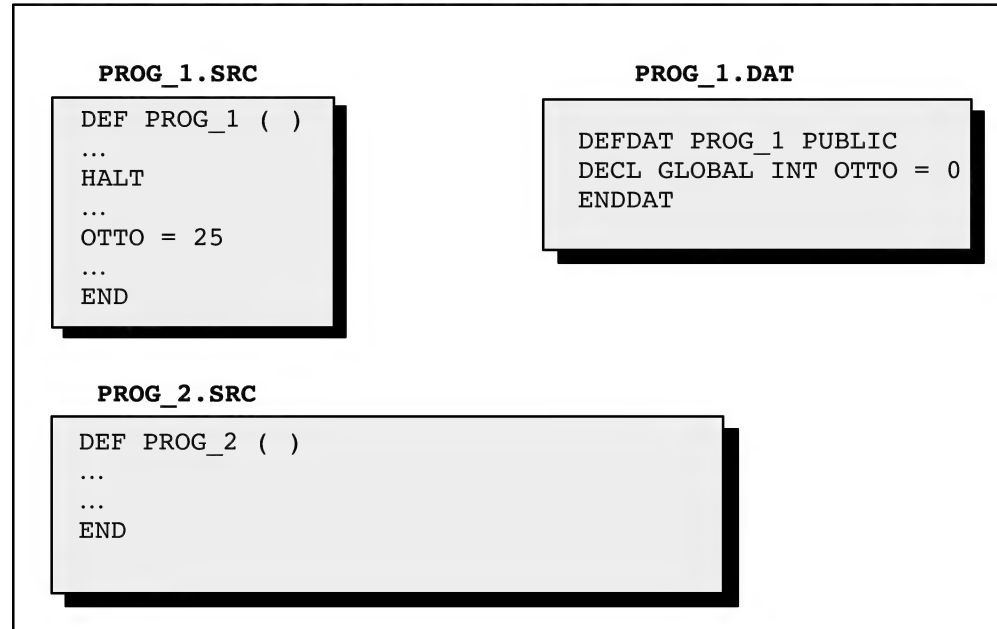


Fig. 49 Importing variables from “foreign” data lists without `Import`



The declaration of a global variable is only permissible in data lists; if it is used in SRC or SUB files, an error message is generated.

Variables, structures, channels and signals which are valid over a long time and are of general significance for a lot of programs can be defined in the predefined, global system data list `$CONFIG.DAT`. Variables in `$CONFIG.DAT` do not need to be declared with `IMPORT` since they are automatically known in all application programs.



Further information on `$CONFIG.DAT` can be found in the chapter **[Variables and declarations]**, section **[System variables and system files]**.

11 External editor

This additional program enhances the robot software by adding a range of functions which are not available in the user interface.

Clean program

Non-referenced motion path points and motion parameters are deleted from the data list.

Setting and offsetting limit switches

Block manipulation

- Select and copy, delete or cut blocks.
- Reverse the path of the robot in the selected area, i.e. the robot now moves to the point previously programmed as the first point in the selected path section last of all and to the point that was programmed as the last point first.
- Reflect the path of the robot in the selected area in the X-Z plane of the world coordinate system.
- Modify the motion parameters (velocity, acceleration, etc.) in the selected area.
- Offset all points within the selected section of the path in the BASE, TOOL or WORLD coordinate system. The offset or rotation can be carried out by manually entering an offset in the respective coordinate system or by teaching reference points.
- Axis-specific offset of all points in the selected area.

Adapt points on the path

- to a different tool coordinate system, or
- to a different base coordinate system

Adapt points on the path

- point coordinates can be offset in the Tool, Base and World coordinate systems while a program is running in the controller.

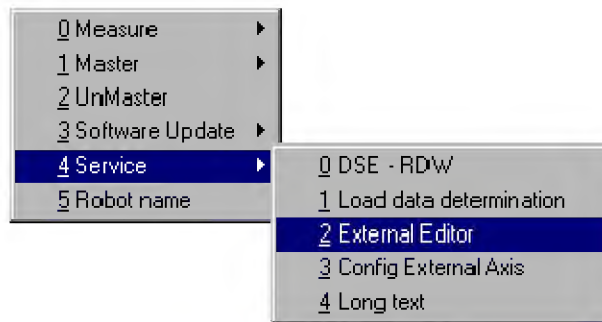
11.1 Starting the external editor



The external editor is not available below the user group "Expert".

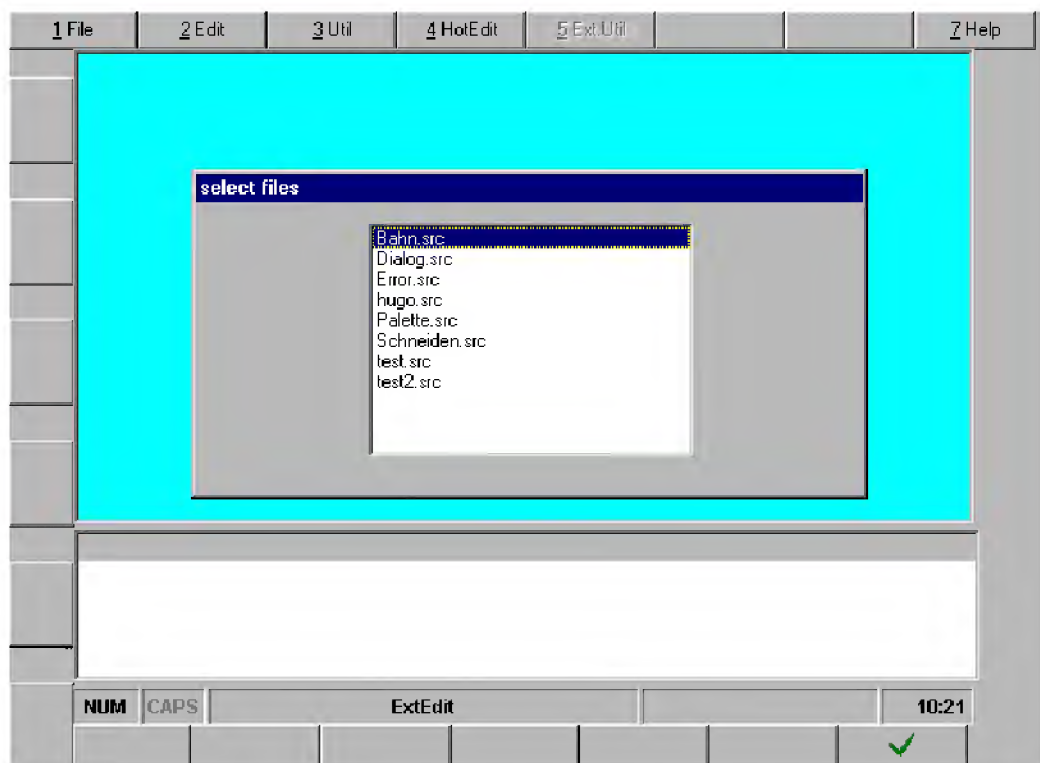
Setup

Activate the external editor via the menu "Setup" and the option "Service" contained in this menu.



If the menu item "Setup" is not available, you must deselect the selected program. Reminder: Do this by selecting the option "Cancel program" from the "Program" menu.

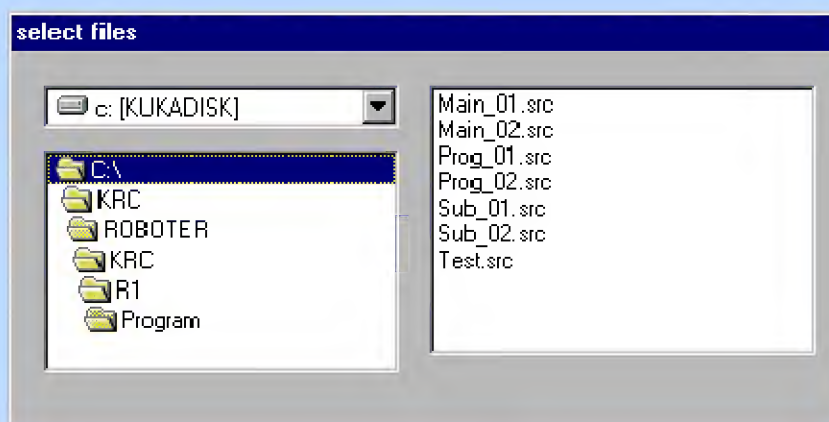
The external editor is then displayed on the screen.



As soon as the editor is started, the "select files" dialog appears. Use the arrow keys "↓" and "↑" to select the file that is to be edited. All SRC files in the directory R1 are available apart from the standard files defined in "HotEdit.ini" (path "C:\Krc\Util\").



If the “external editor” is started as an offline program, selection boxes for specifying the path are also available:

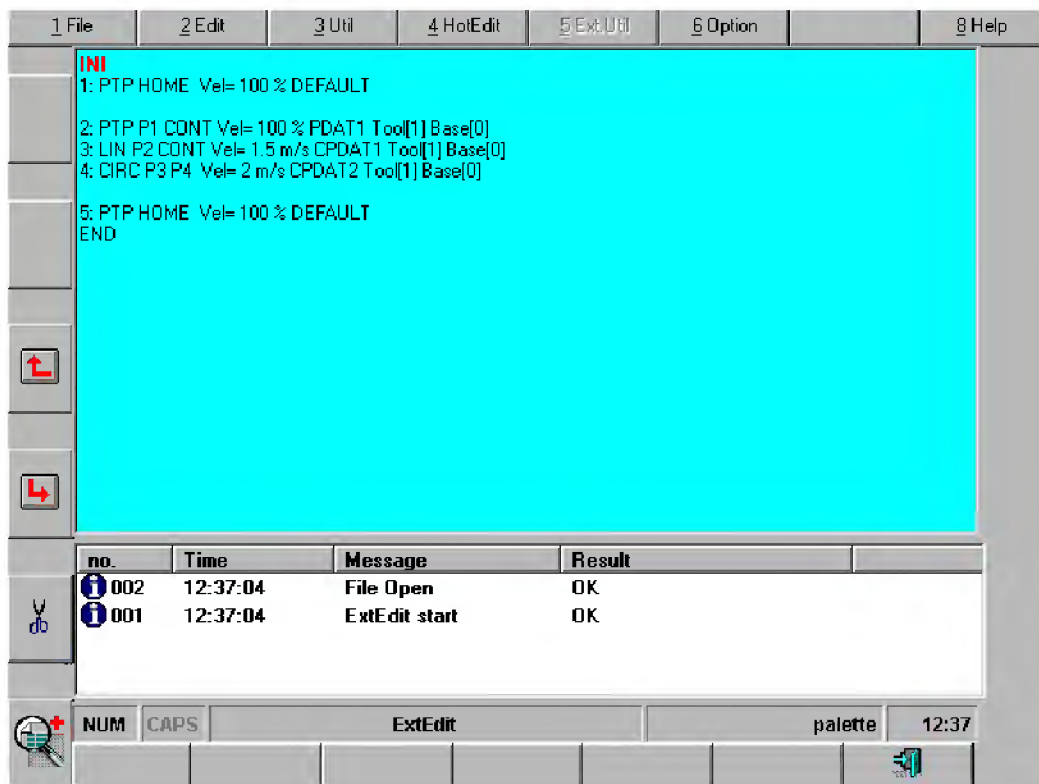


Once you have selected the desired file in this way and pressed the corresponding softkey or Enter, the file is loaded into the editor. The time taken to load the file depends on the size of the file.



If the message “*.DAT file not found” appears, you have attempted to load into the editor a program which does not have a data list. This function is not available in the present version of the editor. Acknowledge this message by pressing the Enter key. Open the “File” menu and select the option “Open file” in order to reach the window “select files”.

When the file has been successfully loaded, the program is displayed in a similar way to that in the editor in the user interface.



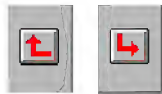
11.2 Operator control

The elements of the normal user interface are also to be found in the external editor. These include menu keys, status keys, softkeys, a program window, a message window and a status line.



Menus can also be opened by holding down the “ALT” key and pressing the letter underlined in the menu key. Menu commands are selected in the same way.

Status keys



Using the status keys “Last” and “Next”, you can move the highlight (focus) line by line towards the start or end of the list. The arrow keys “↑” and “↓” and the keys PGUP and PGDN can be used to carry out the same function.

```
INI
1: PTP HOME Vel= 100 % DEFAULT
2: PTP P1 CONT Vel= 100 % PDAT1 Tool[1] Base[0]
3: LIN P2 CONT Vel= 1.5 m/s CPDAT1 Tool[1] Base[0]
4: CIRC P3 P4 Vel= 2 m/s CPDAT2 Tool[1] Base[0]
5: PTP HOME Vel= 100 % DEFAULT
END
```

Focus

```
INI
1: PTP HOME Vel= 100 % DEFAULT
2: PTP P1 CONT Vel= 100 % PDAT1 Tool[1] Base[0]
3: LIN P2 CONT Vel= 1.5 m/s CPDAT1 Tool[1] Base[0]
4: CIRC P3 P4 Vel= 2 m/s CPDAT2 Tool[1] Base[0]
5: PTP HOME Vel= 100 % DEFAULT
END
```

Focus



The keys “PgUp” and “PgDn” can be used to scroll up or down a page at a time. To do this, it is necessary to disable the “NUM” function or use an external keyboard.

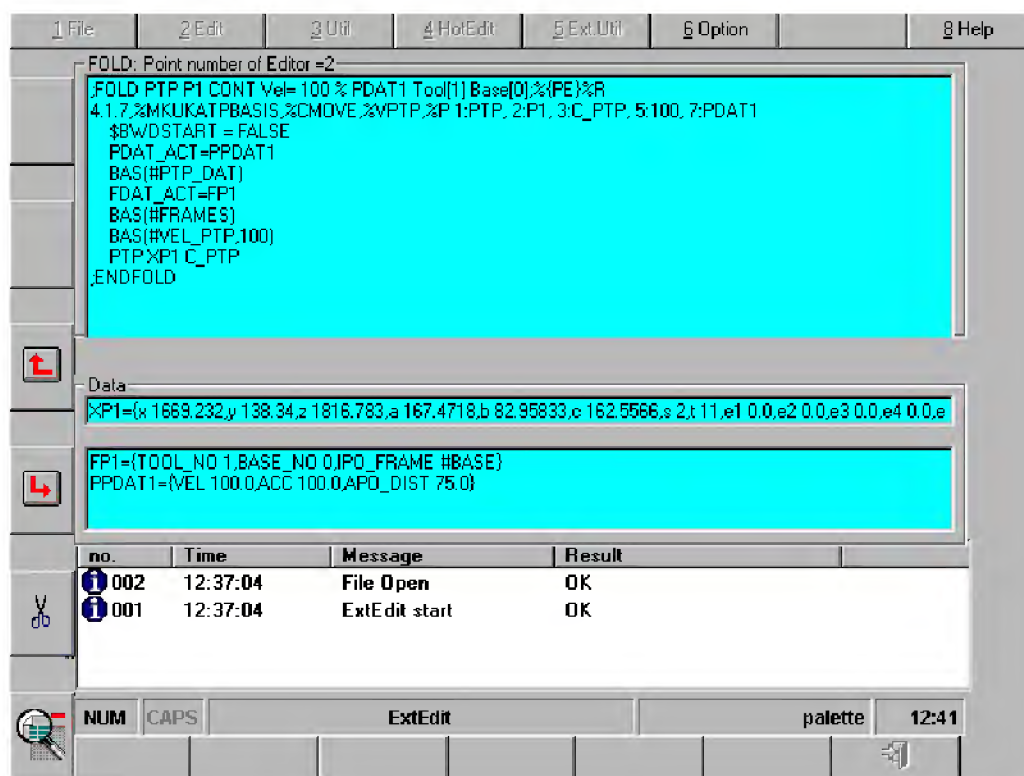


The contents of the message window are deleted.



If you press the status key “KRL”, the program is displayed in the same way as in expert mode with the settings “All FOLDS opn” and “Limited Visibility off”.

The highlighted command is displayed in the section “FOLD” (“SRC” file). The data belonging to the highlighted command are displayed in the section “Data” (“DAT” file).



Pressing the “Zoom –” status key returns you to the display of the entire program.

Softkey bar



This softkey corresponds to the menu command “Close file”, which closes the current program, but leaves the editor running.

Message window

The messages that are relevant for the external editor are displayed in the message window.

no.	Time	Message	Result
002	12:37:04	File Open	OK
001	12:37:04	ExtEdit start	OK

No.: Message type and number

Time: The time the message was generated

Message: The message itself

Result: Gives the result of an operation

Status line

Additional information is displayed in the status line.

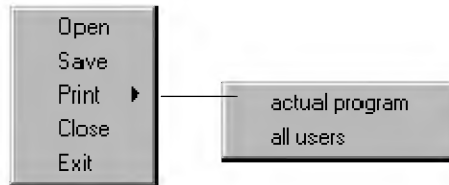
NUM	CAPS	ExtEdit	Palette	12:56
-----	------	---------	---------	-------

- NUM: The numeric keypad is activated for numerical entry
- CAPS: The Caps Lock key is deactivated
- ExtEdit: The "External Editor" module is active
- Palette: The name of the program currently open
- 12:56: The current system time

11.3 “File” menu

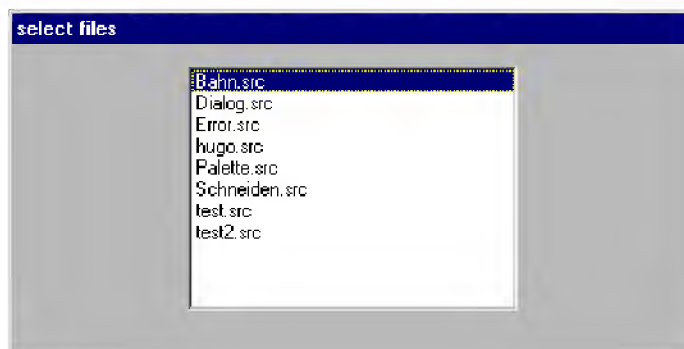


This menu contains a range of functions for working with files.



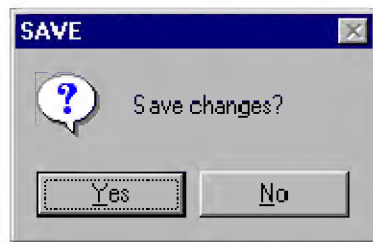
11.3.1 Open

When this option is activated, the window “select files” appears as described in the previous section 11.1.



11.3.2 Save

If the contents of the file loaded in the editor have been changed using the functions available, before saving the new version of the file, or on closing the editor, you are asked if you wish to save the changes.



If you wish to save the changes that have been made, simply press the Enter key to confirm the selected option.



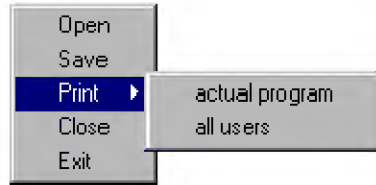
The contents of the existing version of the edited file are then, at least partially, deleted irretrievably.

To cancel the operation, press one of the arrow keys to move the focus to the “No” button. This selection can then be confirmed by pressing the Enter key. The existing version of the edited file then remains unchanged.

11.3.3 Print



This command enables you to print programs to the medium that has been set under “Options” -> “Output setting”.



The following functions are available here:

Current program

The file displayed in the external editor is printed as shown in the editor window.



```
KRC:\Palette.SRC
```

```
*****
```

```
INI
```

```
1:PTP HOME Vel=100 % DEFAULT
```

```
2:PTP P1 CONT Vel=100 % PDAT1 Tool[1] Base[0]
```

```
3:LIN P2 CONT Vel=1.5 m/s CPDAT1 Tool[1] Base[0]
```

```
4:CIRC P3 P4 Vel=2 m/s % CPDAT2 Tool[1] Base[0]
```

```
5:PTP HOME Vel=100 % DEFAULT
```

```
END
```

all users

All application programs are output. If the selected print medium is “File”, the subdirectory “USERPROG” is automatically created.



Depending on the files being printed, printing using the option “all users” can take a long time!

11.3.4 Close file

If the file in the editor has been modified since it was opened, the changes can be saved (as described in Section 11.3.2).

Once the file in the editor has been closed, a different file can be loaded into the editor via the menu item “File” and the option “Open” contained therein (see also Section 11.3.1).

11.3.5 Exit

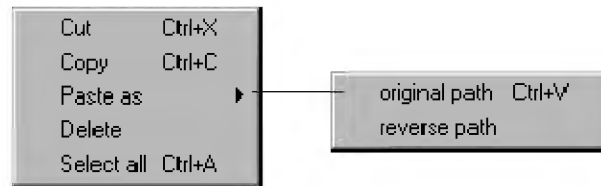
If the file in the editor has been modified since it was opened, the changes can be saved (as described in Section 11.3.2).

The editor window is closed and the robot software user interface reappears.

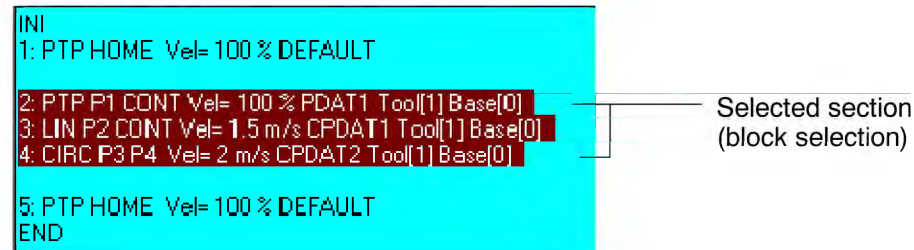
11.4 “Edit” menu



This menu contains a range of block functions.




To select a section, hold down the “SHIFT” key and press one of the two status keys “Last” or “Next” or the corresponding arrow key.



A selected block can be deselected again using the “Escape” key.

11.4.1 Cut (“CTRL”+“X”)

Removes the selected section of the path from the program and copies it to the clipboard ready for subsequent pasting.

11.4.2 Copy (“CTRL”+“C”)

Copies the selected section of the path to the clipboard where it is retained ready for subsequent pasting.

11.4.3 Paste as ...

Original path (“CTRL”+“V”)

Pastes the contents of the clipboard the same way round as they were cut/copied.

Reverse path

Pastes the contents of the clipboard in reverse sequence.

The last point in the selected and cut or copied path section is placed at the start of the pasted block and the first point in the path is placed at the end.

If, for example, the path to a fixture has been taught, this path can be inserted as a reverse path simply by pasting it. In this way the reverse path does not need to be taught separately.

11.4.4 Delete

The selected section of path is deleted without being saved in the clipboard.



The deletion of a selected section of path cannot be undone. If you accidentally delete sections of a program, close the file without saving the changes and then reopen it.

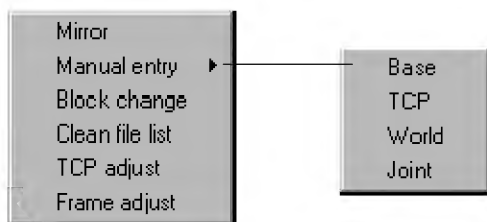
11.4.5 Select all

The whole of the program loaded in the editor is selected.

11.5 “Util” menu



This menu contains options for the geometric manipulation of the program.



You can move the insertion mark between the input boxes using the “↑” and “↓” arrow keys.



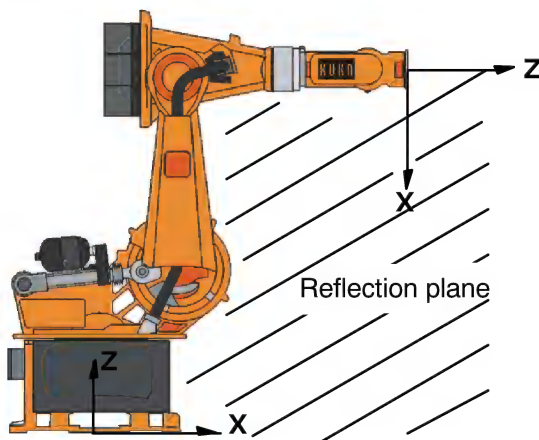
Pressing this softkey saves the values entered.



This softkey is used to terminate the entry without saving the values.

11.5.1 Mirror

This function is used to reflect the positions of the programmed points on the motion path in the X-Z plane of the \$ROBROOT coordinate system. A prerequisite for this is that at least one motion blocks is selected.



Once this option has been selected, a dialog window is opened; in this window you must enter the name of the file in which the loaded program with reflected motion path points is to be saved.



Pressing the Enter key starts the procedure and saves the program with the name specified. This program can then be loaded into the editor.



If you wish to cancel the procedure, press the “Escape” key.

11.5.2 Manual entry

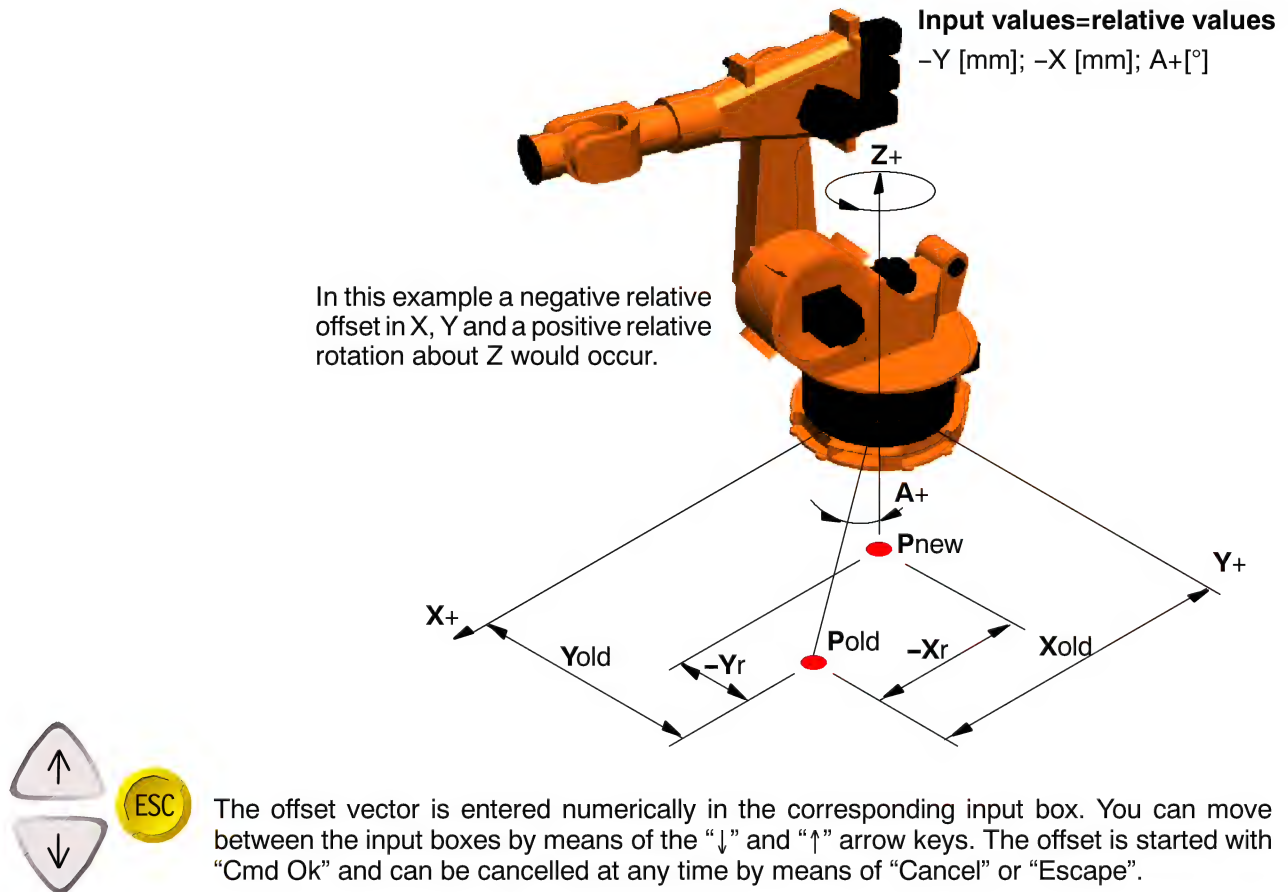
You can use this function to offset the positions of the selected motion path points in:

- the workpiece coordinate system (BASE)
- the TOOL coordinate system (TCP)
- the WORLD coordinate system (WORLD) or
- Joint (axis-specific)

Once one of the possible options has been selected, a window is activated, in which the desired values can be entered. Depending on the coordinate system used, enter either the offset and rotation or the axis angle.

BASE

A Cartesian Base point offset means a relative offset of any point in space in the original coordinate system (WORLD), which is located at the robot base. Cartesian BASE point offset is explained with the help of the example illustrated here.



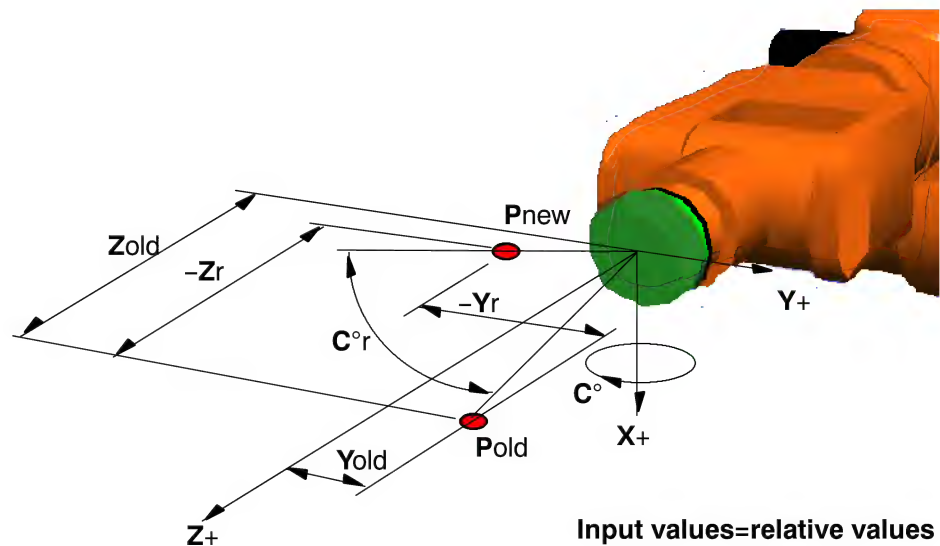
Enter the values for the offset (X, Y, Z) in [mm] and the rotation (A, B, C) in [°] in the input boxes.

Move program: BASE

X=	<input type="text" value="0.0000"/>	[mm]	A=	<input type="text" value="0.0000"/>	[deg.]
Y=	<input type="text" value="0.0000"/>	[mm]	B=	<input type="text" value="0.0000"/>	[deg.]
Z=	<input type="text" value="0.0000"/>	[mm]	C=	<input type="text" value="0.0000"/>	[deg.]

TCP

A Cartesian TCP point offset means a relative offset of any point in space in relation to the TOOL coordinate system. Cartesian TCP point offset is explained with the help of the example illustrated here:



In this example a negative relative offset in Y, Z and a positive relative rotation about X [°] would occur.

A relative offset in X will be disregarded.

The offset vector is entered numerically in the corresponding input box. You can move between the input boxes by means of the “↓” and “↑” arrow keys. The offset is started with “Cmd Ok” and can be cancelled at any time by means of “Cancel” or “Escape”.

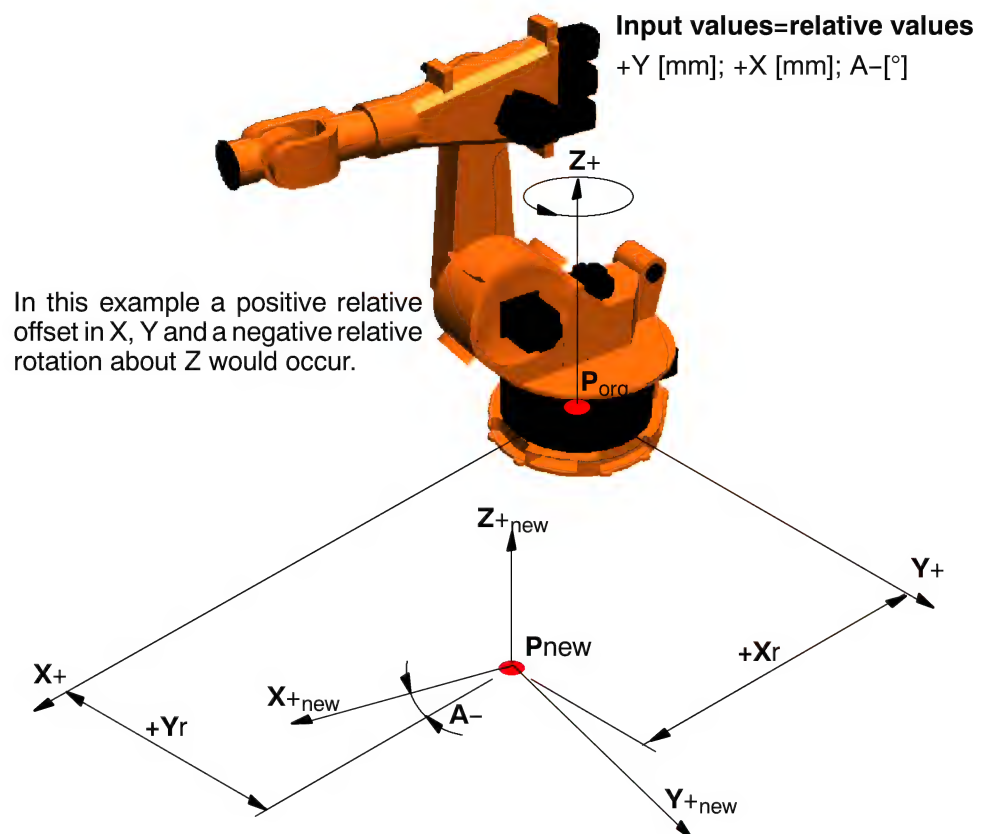
Enter the values for the offset (X, Y, Z) in [mm] and the rotation (A, B, C) in [°] in the input boxes.

Move program: TCP

X=	<input type="text" value="0.0000"/>	[mm]	A=	<input type="text" value="0.0000"/>	[deg.]
Y=	<input type="text" value="0.0000"/>	[mm]	B=	<input type="text" value="0.0000"/>	[deg.]
Z=	<input type="text" value="0.0000"/>	[mm]	C=	<input type="text" value="0.0000"/>	[deg.]

WORLD

A WORLD point offset means a relative offset of the original coordinate system, which is located as standard at the robot base. WORLD point offset is explained with the help of the example illustrated here:



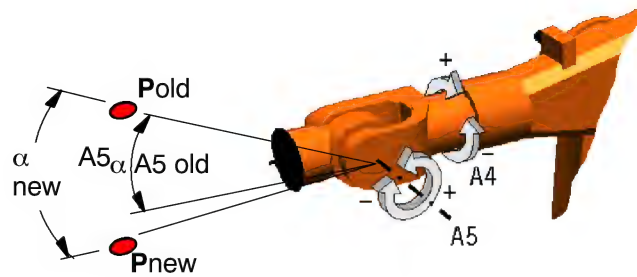
Enter the values for the offset (X, Y, Z) in [mm] and the rotation (A, B, C) in [°] in the input boxes.

Move program: WORLD

X=	<input type="text" value="0.0000"/>	[mm]	A=	<input type="text" value="0.0000"/>	[deg.]
Y=	<input type="text" value="0.0000"/>	[mm]	B=	<input type="text" value="0.0000"/>	[deg.]
Z=	<input type="text" value="0.0000"/>	[mm]	C=	<input type="text" value="0.0000"/>	[deg.]

Axes

An axis-specific point offset occurs when any point in space is reached by a relative motion of the axes. This diagram illustrates a relative offset of axis 5.



In this example, a positive relative rotation of A 5 [°] would occur.

A relative rotation of the other axes will be disregarded.

Input values=relative values
+A5 [°]

The axis rotation is entered in degrees or in increments for the axis concerned. You can move between the input boxes by means of the “↓” and “↑” arrow keys. The offset is started with “Cmd Ok” and can be cancelled at any time by means of “Cancel” or “Escape”.

Enter the values for the axis angle (A1 ... A6) in [°].

Move program: AXIS

A1=	<input type="text" value="0.0000"/>	[deg.]	A4=	<input type="text" value="0.0000"/>	[deg.]
A2=	<input type="text" value="0.0000"/>	[deg.]	A5=	<input type="text" value="0.0000"/>	[deg.]
A3=	<input type="text" value="0.0000"/>	[deg.]	A6=	<input type="text" value="0.0000"/>	[deg.]



Entries can also be made incrementally (E1-E6)[Inc].



Increments = angular momentum from the axis drives.

11.5.3 Block change

Using this function, motion data can be changed in selected program sections. Reminder: Blocks are selected by pressing one of the status keys “Last” or “Next” while holding the “SHIFT” key down.

The window shown below is opened after the function has been called:

no.	Time	Message	Result
004	13:33:34	Change Block	Copy
003	13:33:34	Block change	On
002	13:33:21	File Open	OK
001	13:33:21	ExtEdit start	OK

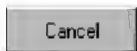
You can move the insertion mark between the input boxes using the “→” and “←” arrow keys.



List boxes are opened using the keyboard shortcut “ALT” + “↓”, or “ALT” + “↑”.



When you have entered the desired changes, press the softkey “Change”.



You can exit the function at any time by pressing the softkey “Cancel”.

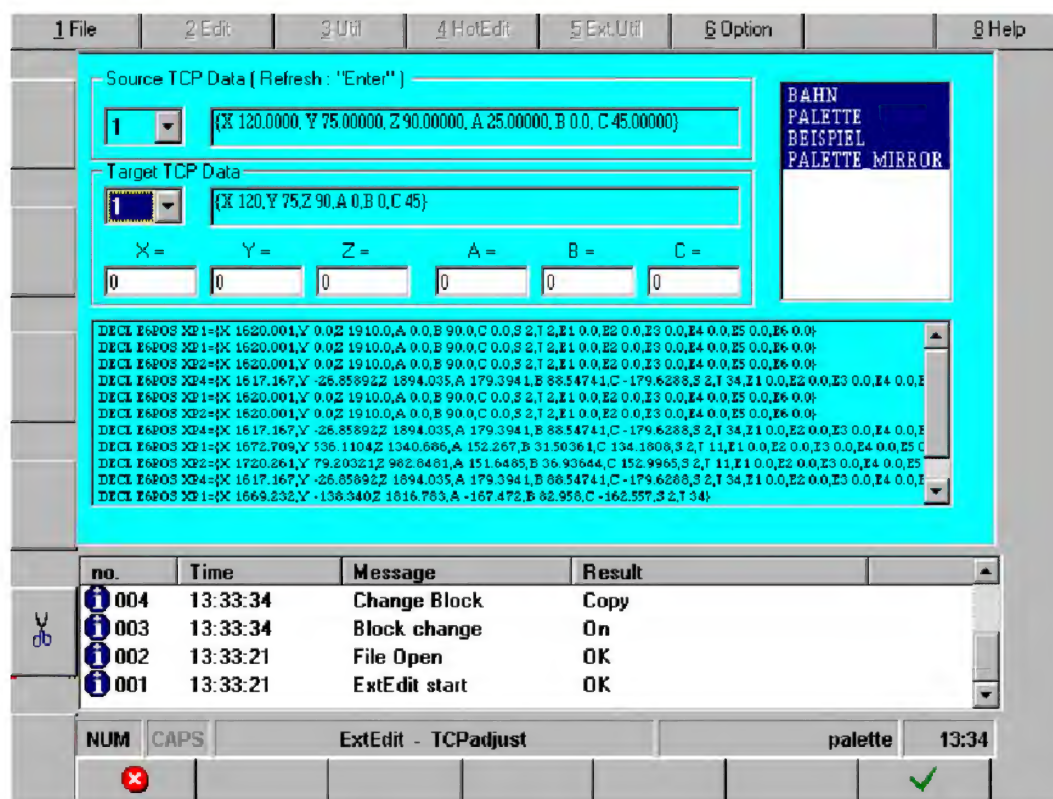
11.5.4 Clean data list

When this function is activated, non-referenced motion path points and motion parameters are deleted from the data list (*.DAT) belonging to the program.

11.5.5 TCP and Frame adjust

You can use this function to adapt the program loaded in the editor to a different TOOL or BASE coordinate system.

The window shown below is opened after the function has been called. TCP adjust has been selected here by way of example.



All programs affected by the adaptation of the TCP or BASE are displayed with their respective points.

In order to be able to move the insertion mark from box to box, the cursor control functions of the numeric keypad must be activated. To do this, press the “NUM” key. You can then move the insertion mark from box to box using the “SHIFT” key.



List boxes are opened using the keyboard shortcut “ALT” + “↓”, or “ALT” + “↑”.

From the list box in the section “Target TCP Data”, select the tool to which the loaded program is to be adapted. The new tool data can also be entered manually in the input boxes “X=”, “Y=”, “Z=”, “A=”, “B=” and “C=”.

When you have entered the desired changes, press the softkey “Change”. You can exit the function at any time by pressing the softkey “End”.

11.6 “HotEdit” menu

4 HotEdit

The menu “HotEdit” allows point coordinates to be offset in the Tool, Base, and World coordinate systems online while a program is running. The menu item “Limits” makes it possible to limit the offset.



11.6.1 Base, TCP and World

Move the cursor to the motion block to be offset or select several motion blocks.

Once one of the options has been selected, a dialog window appears at the bottom of the display; in which the offset (X, Y, Z) and rotation (A, B, C), relative to the respective coordinate system, can be entered. The dialog window depicted here is merely an example and represents all the possible dialog windows.

You can move the insertion mark between the input boxes using the “↑” and “↓” arrow keys.

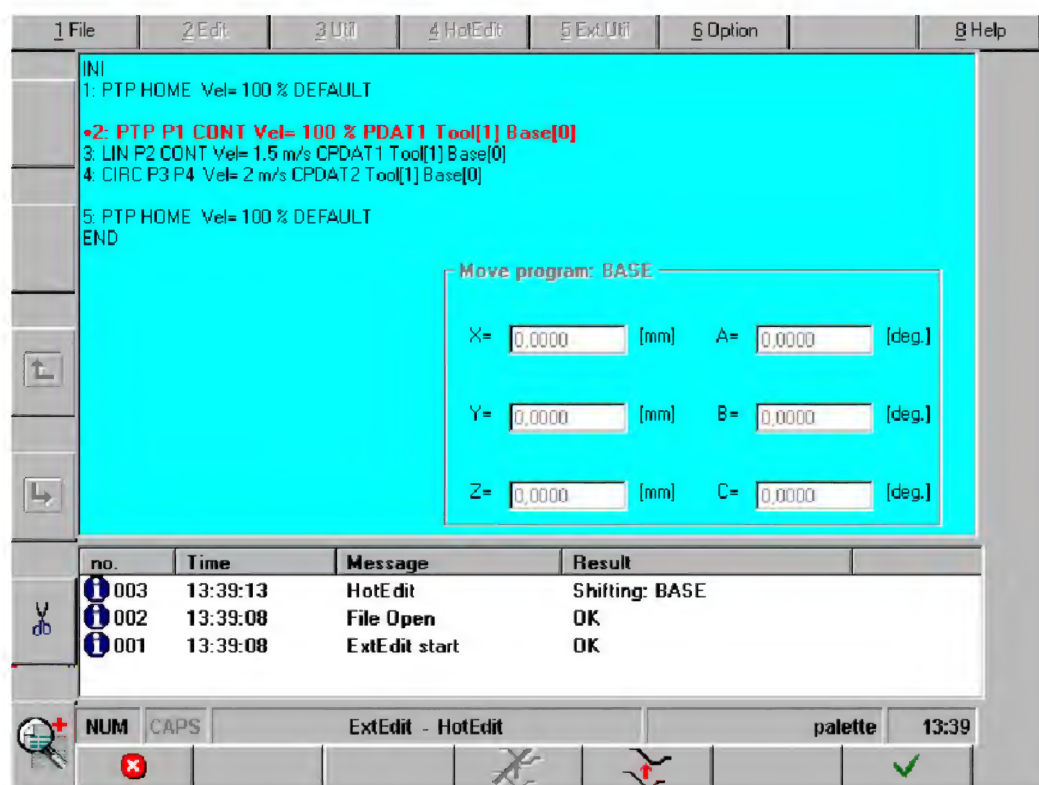


Once you have made all the entries necessary for the offset (integer values), press the softkey “Ok”.



If you wish to cancel the procedure, without executing a program offset, press the softkey “Cancel”.

The dialog window then fades to gray and two additional softkeys are activated.



You can now assign the values entered by pressing the corresponding softkey. The changes to the data are not yet saved and can thus still be undone.



If you change your mind, you can undo the assignment. This is done using the adjacent softkey. This key is only available once the values have been assigned. This function is particularly useful if a program is running and you wish to follow the effect of the offset directly.



If the offset meets your requirements, the modified point coordinates can be saved in the "DAT" file.



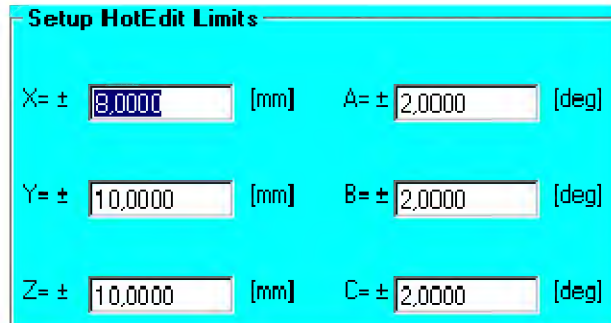
You can use the softkey "Cancel" to cancel the input at any time without saving the changes.



If one of the entered values falls outside the permissible tolerances, a notification or error message appears in the "Move program" window warning that the value exceeds the tolerances.

11.6.2 Limits

When the menu item “Limits” is selected, a window appears in which the offset tolerance limits are entered and can be modified.



To carry out a modification, use the “↑” and “↓” arrow keys to move the cursor to the desired position and enter the new tolerance via the numeric keypad.



Press the softkey “OK” to save the values and terminate the operation.



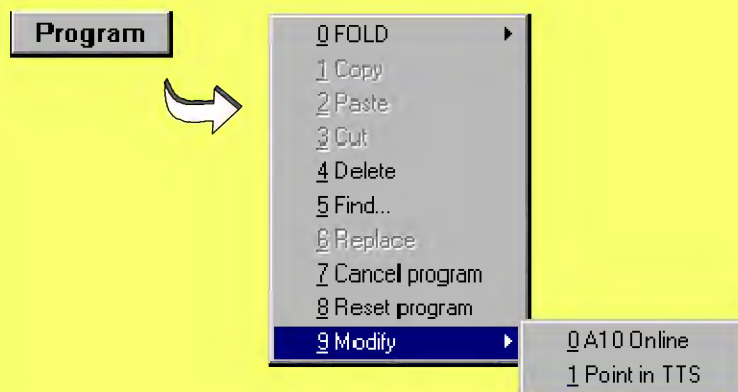
If you wish to terminate the procedure without saving the values, press “Cancel”.

11.6.3 TTS (correction coordinate system)

This function is used for optimizing points in space with the aid of the tool-based moving frame (TTS) and can be used with technology commands (e.g. arc welding or adhesive application). TTS correction is available in the user group "Expert" or above and can be used in all operating modes.



The "Setup" menu is not available if a program is selected. In this case, use the menu command "Program" -> "Modify", which is available at Expert level and higher.



This opens the external editor in which the necessary corrections can be made. Online point correction is only taken into consideration if the advance run pointer or main run pointer has not yet reached the motion command.

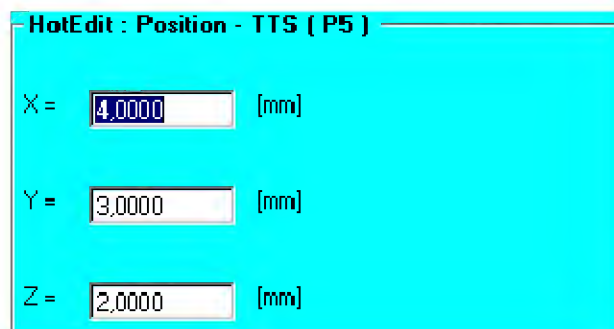
Each time point correction is carried out, a search is made in the local data list (*.DAT) for the necessary points. If one of the required points is not found, an error message is generated.

The search for the point specifying the direction of the path is always carried out starting from the position in the program at which the point to be corrected is situated. An exception is made here for the end point of a weld or adhesive application contour where the search is carried out in the preceding lines.

A change of Base between two motion commands is ignored.

11.6.3.1 Position - TTS

Mark the desired motion command in the external editor and select the command "HotEdit" -> "TTS" -> "Position - TTS". The following window is then opened in the display.



The option "Position - TTS" is only available if a welding or adhesive application motion command is selected.

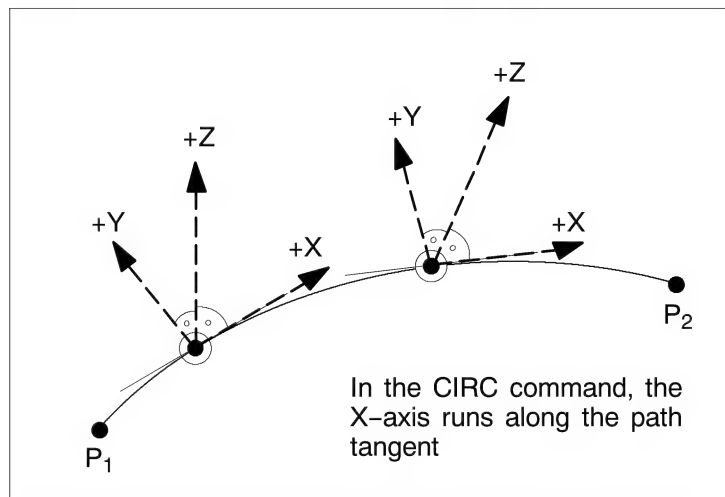
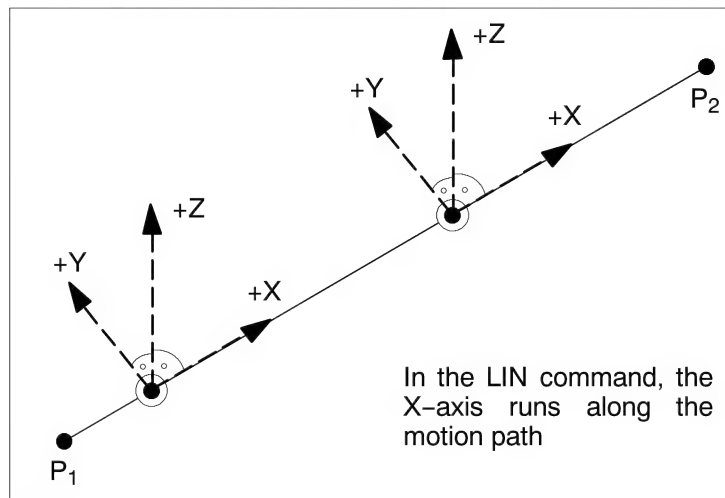


If more than one motion command is selected, the correction is always applied to the first selected command.

Use the “↑” and “↓” arrow keys to move the focus to the desired position. Enter the new values using the keys on the numeric keypad.

X-axis

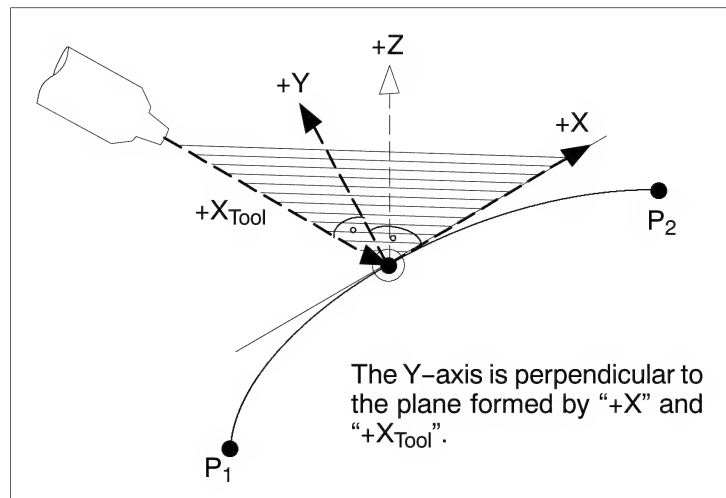
The X-axis in the TTS coordinate system corresponds to a unit vector along the path tangent.



The X-axis of the tool coordinate system may not run parallel to the path tangent as the correction coordinate system cannot otherwise be generated. In this case a corresponding error message will be generated.

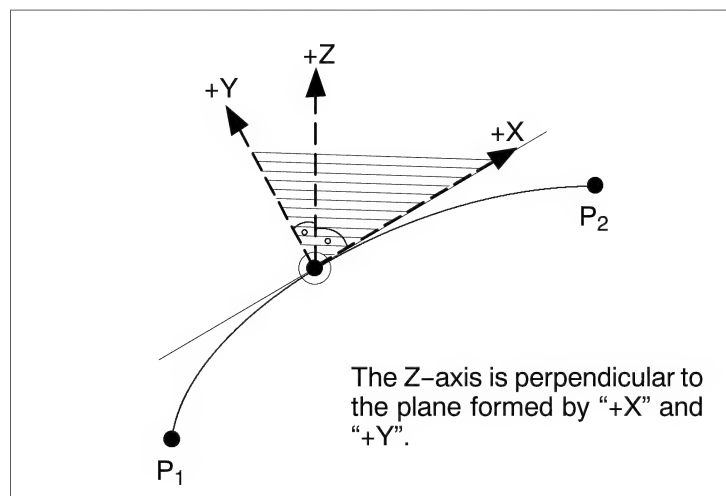
Y-axis

A plane is formed by the unit vector along the path tangent (+X) and the X-axis of the tool coordinate system (+X_{Tool}). The Y-axis is perpendicular to this plane.



Z-axis

A plane is formed by the unit vector along the path tangent (+X) and the Y-axis (+Y). The Z-axis is perpendicular to this plane.



Point correction PTP, LIN



When you have entered the changes in the input boxes, press the softkey "OK".



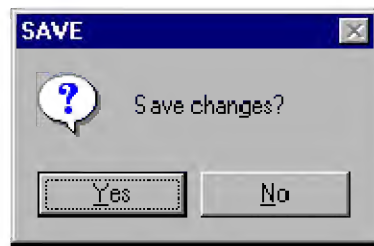
You can now assign the values entered by pressing the corresponding softkey. The changes to the data are not yet saved and can thus still be undone.



If you change your mind, you can undo the assignment. This is done using the adjacent softkey. This key is only available once the values have been assigned.



Press the softkey "OK" again and confirm the subsequent request for confirmation to save the changes.



You can use the softkey "Cancel" to cancel the input at any time.

Point correction CIRC



Once you have entered the changes, assign the corrections to the auxiliary point (Aux).



With this softkey, select the end point (End) once you have entered the changes in the input boxes.



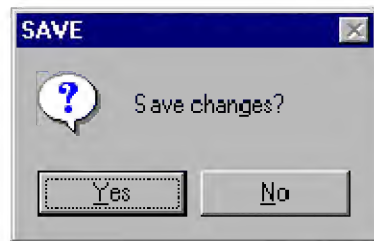
You can now assign the values entered by pressing the corresponding softkey. The changes to the data are not yet saved and can thus still be undone.



If you change your mind, you can undo the assignment. This is done using the adjacent softkey. This key is only available once the values have been assigned.



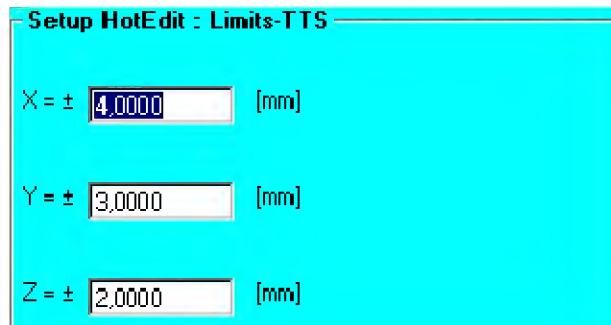
If the offset meets your requirements, the modified point coordinates can be saved by confirming the subsequent request for confirmation.



You can use the softkey "Cancel" to cancel the input at any time without saving the changes.

11.6.3.2 Limits-TTS

Here you can define the limit values of the TTS correction. These limit values are automatically monitored during point correction.



The maximum permitted tolerances are defined in the file "C:\KRC\Util\HotEdit.ini". If a larger value is entered, an error message is generated.



Limit values greater than 5 mm for TTS correction are not sensible.

Use the "↑" and "↓" arrow keys to move the focus to the desired position. Enter the new tolerance using the keys on the numeric keypad.



Press the softkey "OK" to save the tolerances.

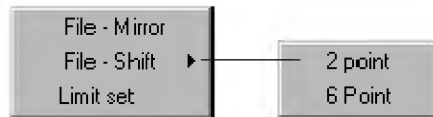


If you do not wish to save the values, press "Cancel".

11.7 “ExtUtil” menu



This menu contains mirror and offset functions and functions for setting the software limit switches.

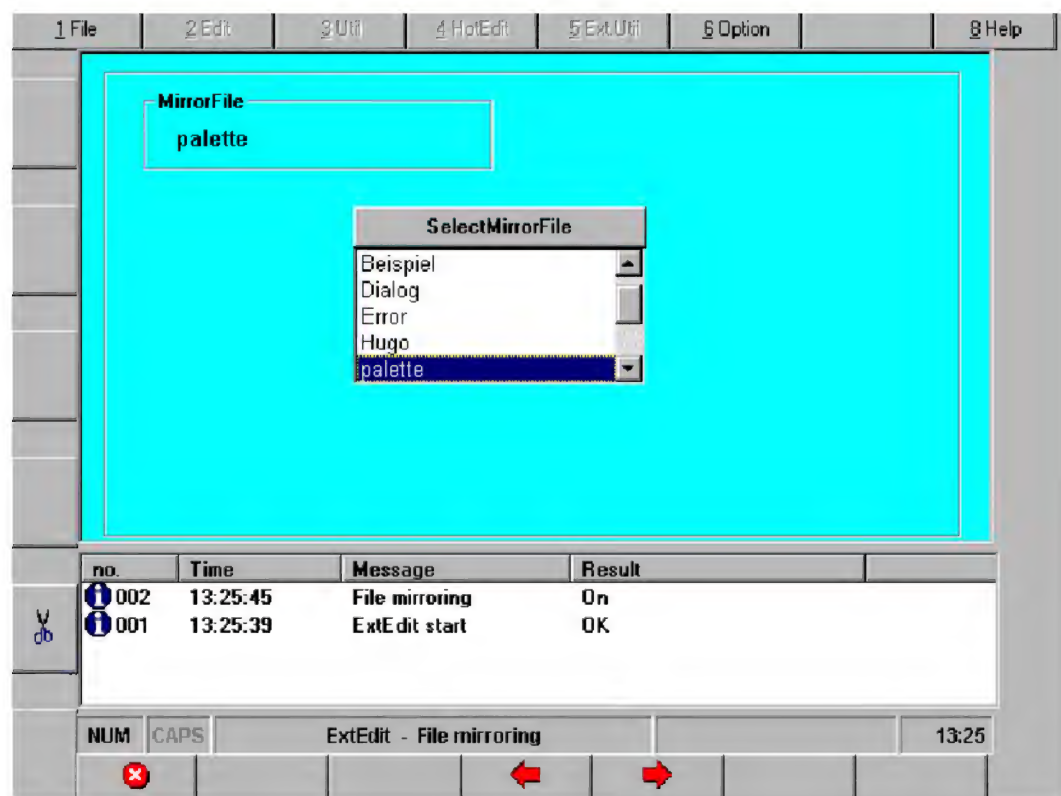


It is only available if no program has been loaded into the external editor.

11.7.1 File – Mirror

Unlike the “Mirror” function (Section 11.5.1), which is used to mirror selected motion path points in selected motion blocks, the “File – Mirror” function makes it possible to mirror entire motion programs.

After starting the function, specify the source file to be mirrored.



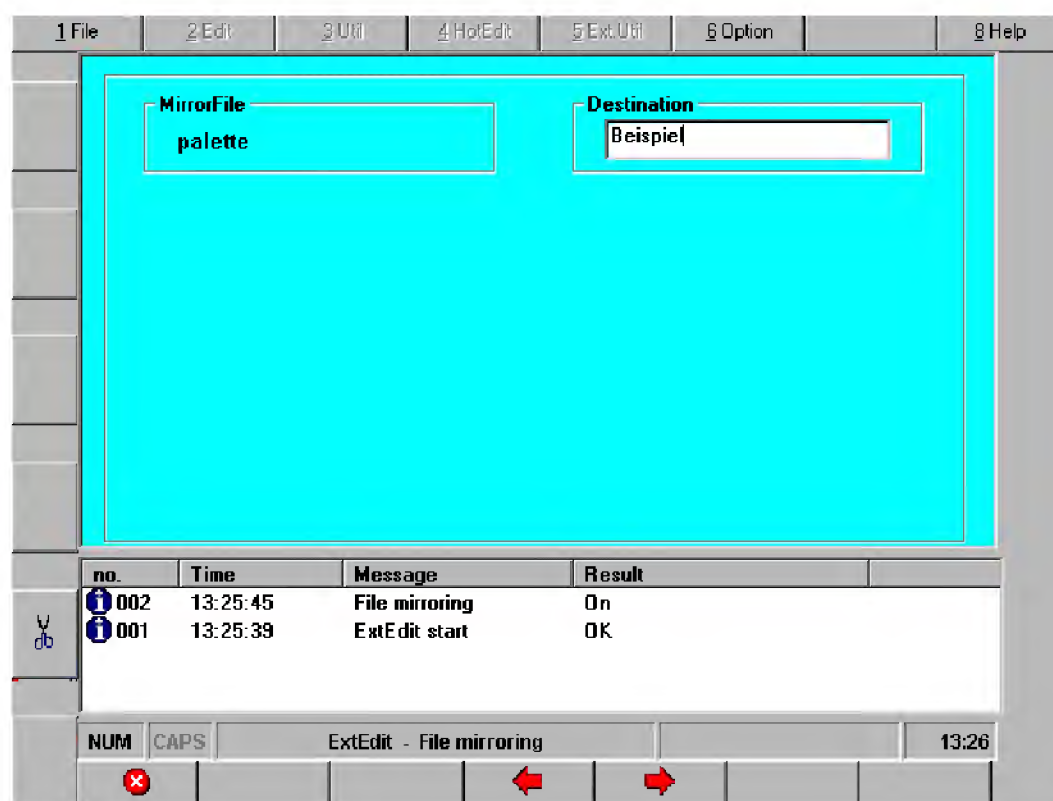
Select a file to mirror using the “↑” and “↓” arrow keys.



Once you have selected the desired file, press the “Right arrow” softkey.



The function can be terminated at any time by means of the softkey “Cancel”.



Enter a name for the mirrored file, with a maximum length of 8 characters.



This softkey takes you back to the previous page, where you can select a different source file.



Once you have entered the desired file name, press this softkey.

no.	Time	Message	Result
002	13:25:45	File mirroring	On
001	13:25:39	ExtEdit start	OK

It is now possible to enter a comment which will then be displayed in the Navigator.



This softkey takes you back to the previous page where you can enter the name of the (mirrored) file to be created.



Once you have entered the desired comment, press this softkey.

The corresponding program is then created. You can now scroll backwards using the “Left arrow” key, in order, for example, to create another program or change the comment.



It is also possible to exit the “File – Mirror” function. This is done by pressing the softkey “Close”.

11.7.2 File – Shift

This function allows you to offset the positions of the selected motion path points using a vector (2 point), or simultaneously offset the points and rotate the reference coordinate system (6 point).

2-point offset

For the 2-point offset function, a reference file (any name; .src, .dat) must be taught. Two points are stored in this reference file. These points define the offset vector (X, Y, C) for the offset of the selected motion path points.

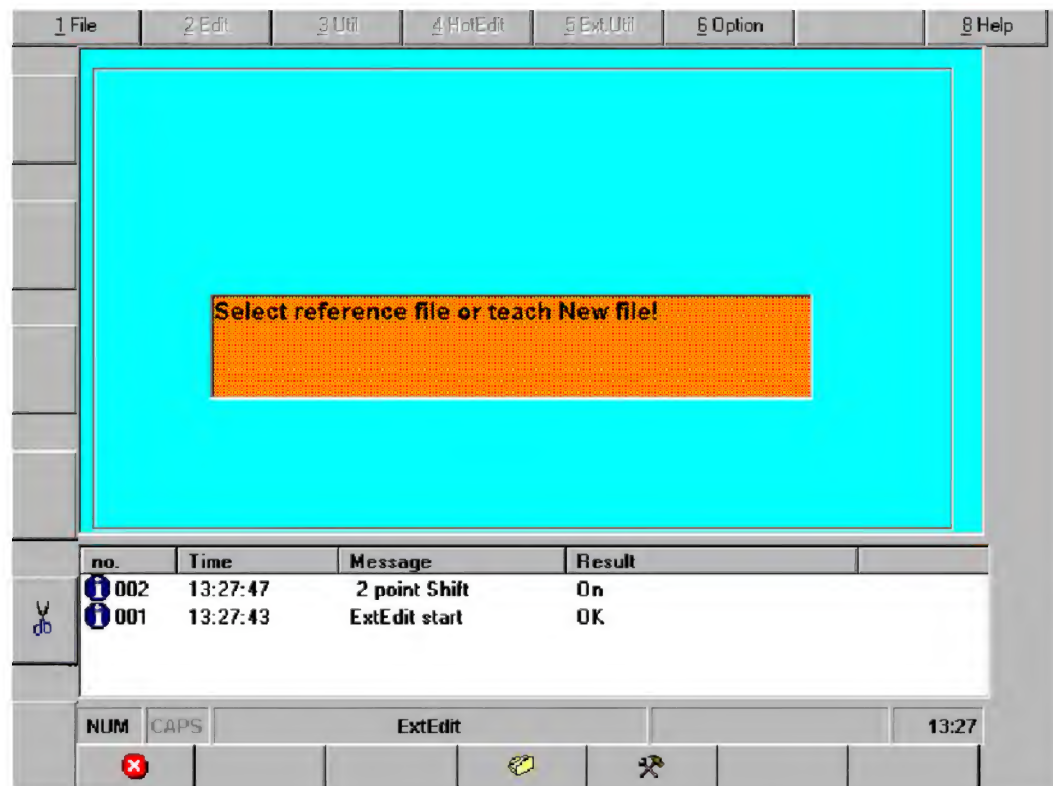
6-point offset

The 6-point offset function works using the same principle. A total of six points must be taught in the corresponding reference file. The first three points define the source base, the last three define a destination base. Identical points must be taught for both the source and the destination. These values are used to calculate the bases. The offset and rotation between the source and destination bases define the amount by which the selected points are offset.



The vector must already have been saved in a file before the function is activated.
The motion blocks to be offset must also be selected.

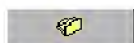
Once the option 2 point or 6 point offset has been selected, the program requires the specification of the corresponding reference file.



The function can be terminated at any time by means of the softkey "Cancel".

You now have the following options:

11.7.2.1 Use existing reference file



This softkey can be used to specify a reference file and a file to be offset.

The screenshot shows the ExtEdit interface with a menu bar (1 File, 2 Edit, 3 Util, 4 HotEdit, 5 ExtUtil, 6 Option, 8 Help) and a toolbar. The main area has a 'ReferenceFile' field containing 'Beispiel'. A 'SelectRefFile' dialog box is open, showing a list of files: 'bahn', 'Beispiel' (highlighted), 'Dialog', 'Error', and 'Hugo'. Below the main area is a table with the following data:

no.	Time	Message	Result
002	13:27:47	2 point Shift	On
001	13:27:43	ExtEdit start	OK

At the bottom, there is a 'NUM' field, a 'CAPS' field, and a 'Right arrow' softkey.



First select the reference file using the arrow keys, then press the “Right arrow” softkey.



This softkey takes you back to the previous page.

The screenshot shows the ExtEdit interface with a menu bar (1 File, 2 Edit, 3 Util, 4 HotEdit, 5 ExtUtil, 6 Option, 8 Help) and a toolbar. The main area has a 'FileToShift' field containing 'bahn' and a 'ReferenceFile' field containing 'Beispiel'. A 'SelectFileToShift' dialog box is open, showing a list of files: 'bahn' (highlighted), 'Beispiel', 'Dialog', 'Error', and 'Hugo'. Below the main area is a table with the following data:

no.	Time	Message	Result
002	13:27:47	2 point Shift	On
001	13:27:43	ExtEdit start	OK

At the bottom, there is a 'NUM' field, a 'CAPS' field, and a 'Right arrow' softkey.

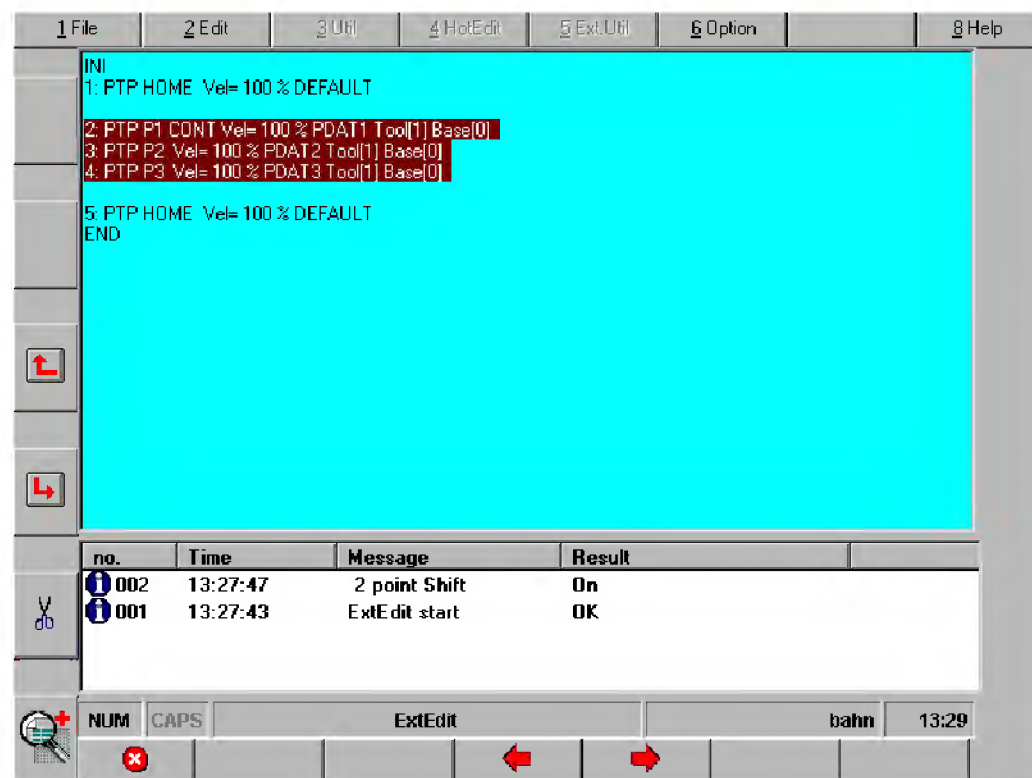


Use the arrow keys again to select the file to be offset. Then press the softkey “Right arrow” again.



This softkey takes you back to the previous page.

The program to be offset can then be loaded into the editor. Select here the motion commands to be offset.



Press the “Right arrow” softkey again. The selected points are then offset in the file. Progress is shown by means of a progress indicator bar in the message window.



This softkey takes you back to the previous page.

The result of the operation is displayed in the message window.

no.	Time	Message	Result
003	13:29:28	2 point Shift	Position XP1 to XP3 shifted
002	13:27:47	2 point Shift	On
001	13:27:43	ExtEdit start	OK

11.7.2.2 Create new reference file



This softkey is used to create a new reference file. This new reference program is then selected ready for the operator to teach the necessary points.

```
1 INI
2
3 ; Teach here the first position
4
5 ; Teach here the second position
6
```

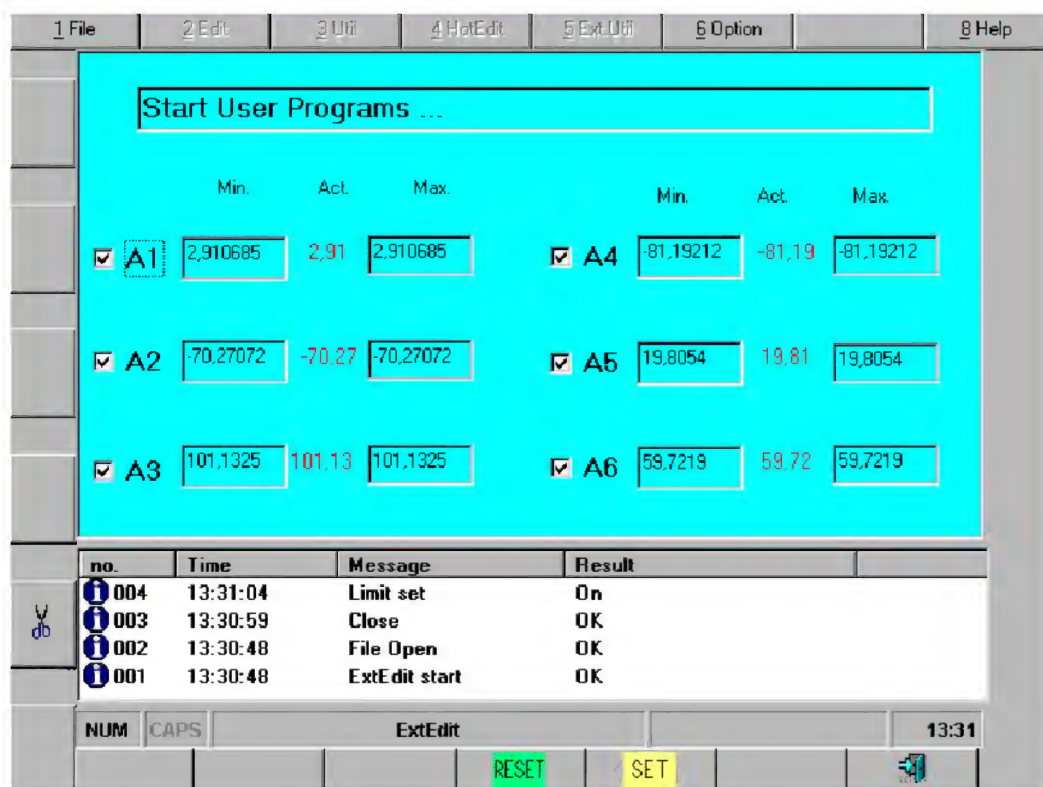
/R1/REF_2POINT.SRC Ln 1, Col 0

After teaching the points, deselect the program and restart the external editor. Use the procedure described in Section 11.7.2.1.

11.7.3 Setting the software limit switches

The software limit switches of the individual axes can be adapted to the motion programs using this function.

To do this, call the “Set software limit switches” function and then switch back to the robot controller user interface by pressing the window selection key. Now start the motion programs whose axis values are to be monitored in the background. Once all the relevant programs have been executed, call the function again. The values thus calculated can be set as software limit switches.



	Min.	Act.	Max.		Min.	Act.	Max.
<input checked="" type="checkbox"/> A1	2,910685	2,91	2,910685	<input checked="" type="checkbox"/> A4	-81,19212	-81,19	-81,19212
<input checked="" type="checkbox"/> A2	-70,27072	-70,27	-70,27072	<input checked="" type="checkbox"/> A5	19,8054	19,81	19,8054
<input checked="" type="checkbox"/> A3	101,1325	101,13	101,1325	<input checked="" type="checkbox"/> A6	59,7219	59,72	59,7219

no.	Time	Message	Result
004	13:31:04	Limit set	On
003	13:30:59	Close	OK
002	13:30:48	File Open	OK
001	13:30:48	ExtEdit start	OK

NUM CAPS ExtEdit 13:31

RESET SET



The axis positions are monitored in the background for as long as the “Set software limit switches” function remains active.

The minimum/maximum values that arise and the current axis positions of axes 1 to 6 are entered in the corresponding boxes.



The axes for which the limit switches are to be set can be indicated by means of a check box. Use the arrow keys to select the desired axes. The individual axes can be activated or deactivated using the space-bar. A check sign means that the software limit switches can be set for this axis.



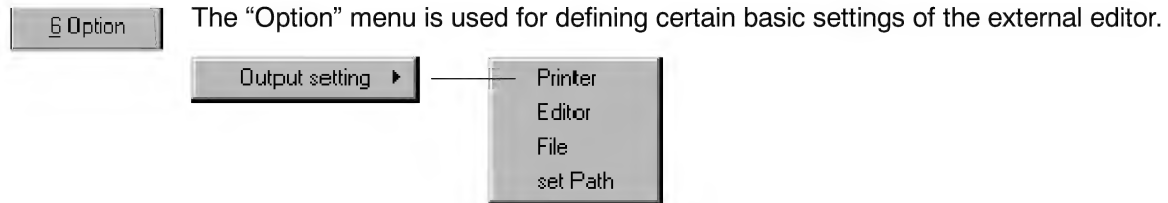
The software limit switches can be reset to their original values by pressing “Reset”.

Pressing the softkey “Set” sets the software limit switches of the selected axes with their new values. This action can be undone, at any time, by pressing the softkey “Reset”.



A buffer of 5° is added to the measured axis values.

11.8 “Option” menu



11.8.1 Output setting

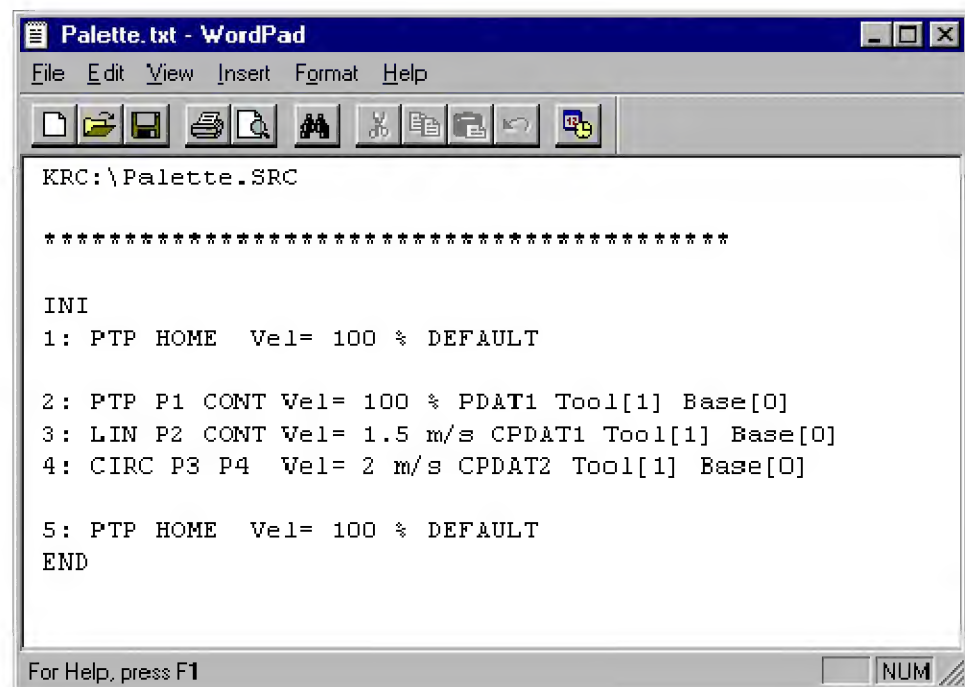
This menu can be used to set the desired print medium which will be used in the case of “Print” -> “actual program” or “Print” -> “all users”.

Printer

If a printer has been configured for the operating system, this will be used for the output. If, on the other hand, no printer has been installed, a corresponding message is displayed in the message window.

Editor

The selected program is loaded for editing in the “WordPad” editor provided with the operating system.



File

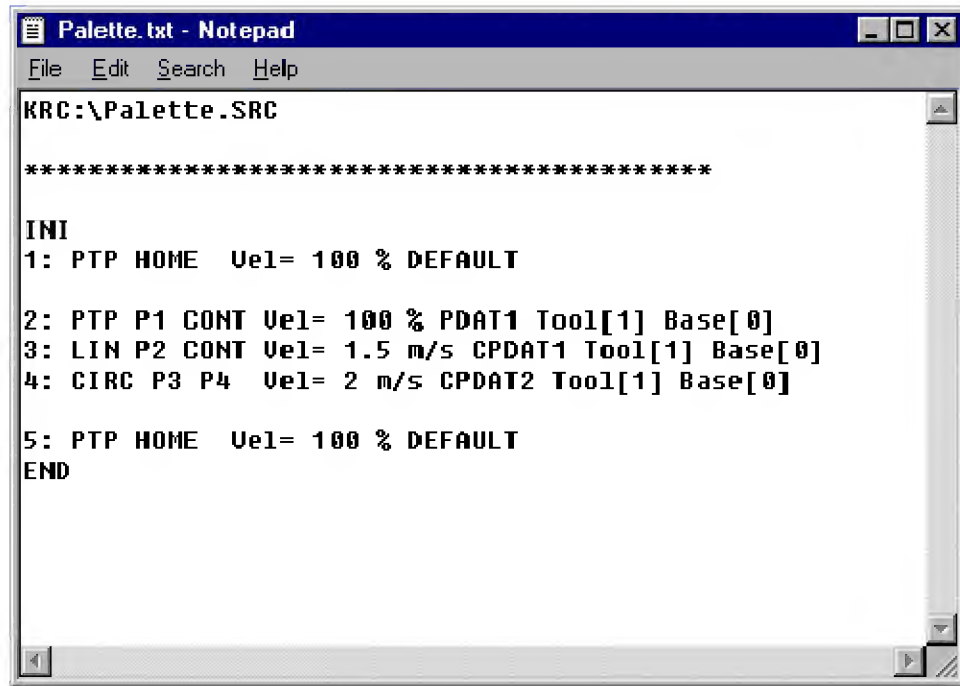
If the current program is output by selecting the command “Print”, it is saved as a text file under the path or file name set in “set Path”.

If all application programs are output via the command “Print”, the subdirectory “USER-PROG” is created in the folder defined in “set Path” and the programs are saved here as text files.



The default setting for “set Path” is “C:\”.

Each of the saved programs can be viewed or modified using a text editor.



```

KRC:\Palette.SRC

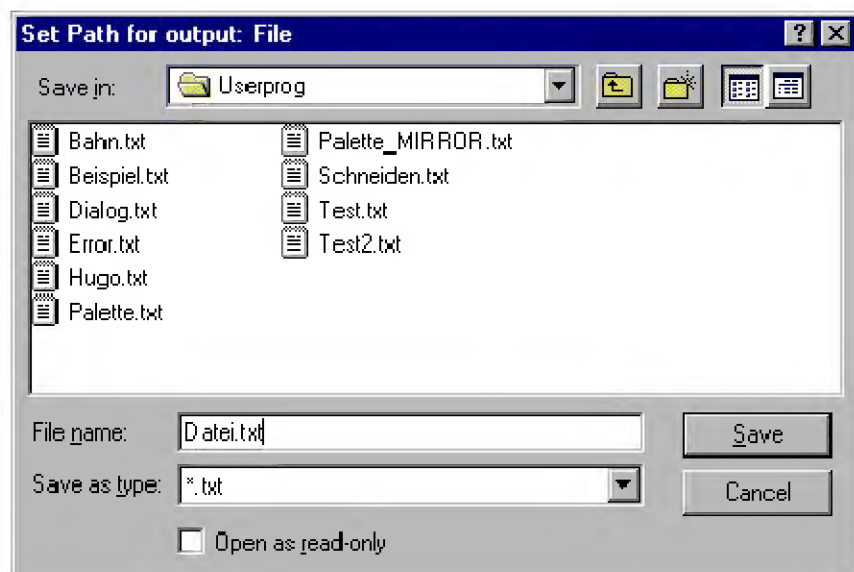
*****

INI
1: PTP HOME Vel= 100 % DEFAULT
2: PTP P1 CONT Vel= 100 % PDAT1 Tool[1] Base[0]
3: LIN P2 CONT Vel= 1.5 m/s CPDAT1 Tool[1] Base[0]
4: CIRC P3 P4 Vel= 2 m/s CPDAT2 Tool[1] Base[0]
5: PTP HOME Vel= 100 % DEFAULT
END

```

set Path

This option is only necessary if programs are to be saved to the hard drive as files. The target directory and the file name of the file to be printed can be specified in the dialog window. If all application programs are output using the menu option “File”, the subdirectory “USER-PROG” is created and the text files are saved here.



11.9 “Help” menu



This menu contains version information and a display option.



11.9.1 Version

The version number of the external editor is displayed in the message window.

no.	Time	Message	Result	
i 003	09:10:35	Version	4.1.14	Version number
i 002	09:10:31	File Open	OK	
i 001	09:10:31	ExtEdit start	OK	

11.9.2 Stay on top

Once this option has been selected, the robot software user interface will no longer come to the foreground until the editor has been closed.



NOTES:

Symbols

.ERR, 12
 !, 106
 ! sign, 103
 ?, 107
 :, 108
 # sign, 41
 #BASE, 65, 79
 #CONSTANT, 78
 #INITMOV, 78, 79
 #PATH, 79
 #TCP, 65
 #VAR, 78
 \$ sign, 55
 \$ACC.CP, 77
 \$ACC.ORI1, 77
 \$ACC.ORI2, 77
 \$ACC_AXIS, 66
 \$ADVANCE, 86
 \$ALARM_STOP, 55, 56
 \$APO.CDIS, 93, 96
 \$APO.CORI, 93, 96
 \$APO.CPTP, 90
 \$APO.CVEL, 93, 96
 \$APO_DIS_PTP, 90
 \$BASE, 63, 70
 \$CIRC_TYPE, 79, 82
 \$CONFIG.DAT, 8, 57
 \$CUSTOM.DAT, 56
 \$CYCFLAG, 55
 \$FLAG, 54
 \$IBUS_ON, 56
 \$IPO_MODE, 65
 \$MACHINE.DAT, 8, 56
 \$NULLFRAME, 70
 \$NUM_AX, 56
 \$ORI_TYPE, 78, 82
 \$OUT_C, 133
 \$POS_ACT, 64, 65
 \$PRO_MODE, 22
 \$PSER, 56
 \$ROBCOR.DAT, 8, 57
 \$ROBROOT, 63, 70
 \$SET_IO_SIZE, 133
 \$TIMER, 54
 \$TIMER_FLAG, 54
 \$TIMER_STOP, 54
 \$TOOL, 63, 70

\$VEL.CP, 77
 \$VEL.ORI1, 77
 \$VEL.ORI2, 77
 \$VEL_AXIS, 66
 \$WORLD, 63, 70

Numbers

2D arrays, 36
 3D arrays, 37

A

A10.DAT, 8
 A10.SRC, 8
 A10_INI.DAT, 8
 A10_INI.SRC, 8
 A20.DAT, 8
 A20.SRC, 8
 A50.DAT, 8
 A50.SRC, 8
 ABS(X), 52
 Absolute value, 52
 Acceleration, 77
 ACOS(x), 52
 Advance run, 86
 Advance run stop, 87, 127
 Aggregate, 38
 All FOLDS cls, 19
 All FOLDS opn, 19
 Altering programs, 14
 Ambiguities, 74
 Ambiguous robot kinematics, 74
 Analog inputs, 141
 Analog outputs, 138
 ANDing, 49
 ANIN, 141
 ANIN OFF, 141
 ANIN ON, 141
 ANIN/ANOUT, 138
 ANOUT OFF, 138
 ANOUT ON, 138
 Approximate positioning, 89
 Approximate positioning contour, 89
 Arc cosine, 52
 Arc sine, 52
 Arc tangent, 52
 ARCSPS.SUB, 8
 Arithmetic operators, 42
 Array index, 35

Arrays, 35
ATAN2(Y,X), 52
Automatic advance run stop, 87
AXIS, 40, 60, 107
Axis acceleration, 68
Axis velocity, 68

B

B_AND, 49, 51
B_EXOR, 49, 51
B_NOT, 49, 51
B_OR, 49, 51
BAS.SRC, 8, 72
BASE, 65
Base coordinate system, 64
Base-related interpolation, 64
Basic, 74
BCO, 73
Bin Dec, 34
Binary inputs/outputs, 128
Binary system, 33
Bit operators, 49
Block change, 195
Block coincidence, 73
Block copy, 14
Block cut, 15
Block functions, 14
Block identifier, 117
Block paste, 15
BOOL, 33, 34
BOSCH.SRC, 8
BRAKE, 160

C

C_DIS, 93, 96, 109
C_ORI, 93, 96, 109
C_PTP, 90, 109
C_VEL, 93, 96, 109
CA, 84
Cartesian coordinate transformation, 60
CELL.DAT, 8
CELL.SRC, 8, 9
CHAR, 33, 35
Character strings, 38
CIRC, 84, 113
CIRC !, 103
CIRC-CIRC approximate positioning, 96

CIRC-LIN approximate positioning, 96
CIRC_REL, 84, 113
Circular angle, 84
Circular motions, 84
Clean data list, 195
Comments, 27
Compiler, 12
Compiling, 12
Computer advance run, 86
Conditional branch, 116
CONFIRM, 126
Constant + path-related, 79
Constant + space-related, 81
CONTINUE, 88
Continuous Path, 77
Continuous-path motions, 77
Coordinate system, 70
Coordinate systems, 59
Coordinate transformation, 60
Correction coordinate system, 200
COS(X), 52
Counting loop, 118
CP motions, 77
CP-PTP approximate positioning, 100
Creating a new program, 11
Creating and editing programs, 11
CSTEP, 22
CTRL-C, 14
CTRL-V, 15
CTRL-X, 15
Current FOLD opn/cls, 19
Cyclical flags, 55

D

Data list, 10
Data lists, 175
Data manipulation, 42
Data objects, 31
Data type, 31
Decimal system, 33
DECL, 31
Declaration, 10
Declaration section, 9
DEF, 9, 145
DEFDAT, 175
DEFFCT, 146
Delete, 15
DIGIN, 143

DIGIN OFF, 143
DIGIN ON, 143
Digital inputs, 143
Digital inputs/outputs, 131
Disable/Enable, 156
DISTANCE, 166
Distance criterion, 93

E

E6AXIS, 40, 107
E6POS, 107
Edge-triggered, 156
Edit cursor, 105
Editing, 12
Editor, 14
ELSE, 116
END, 9
ENDFOLD, 19
ENDFOR, 118
Endless loop, 123
ENDLOOP, 123
ENDWHILE, 120
ENUM, 40
Enumeration types, 40
Error treatment, 24
Exclusive ORing, 49
EXIT, 123
EXT, 72
External editor, 179

F

Fast measurement, 163
File concept, 9
File list, 10
File structure, 9
Filtering out bits, 50
Flags, 55, 164
FLT_SERV.DAT, 8
FLT_SERV.SRC, 8
FOLD, 19
FOR, 118
Forward transformation, 60
FRAME, 40, 107
Frame adjust, 196
Frame linkage, 43
Functions, 9, 145

G

Geometric data types, 40
Geometric operator, 43
Geometric operator ":", 108
GLOBAL, 177
Global, 146
Global data lists, 176
Global variable, 177
GO, 22
GOTO, 115
Gripper-related interpolation, 65

H

H50.SRC, 8
H70.SRC, 8
HALT, 125
Hex Dec, 34
Hexadecimal system, 33
Hiding program sections, 19
Higher motion profile, 67
Home run, 73

I

IF, 116
Index, 35
INI, 72
Input/output instructions, 127
Instruction, 10
Instruction section, 9
INT, 33
INTERRUPT, 154
Interrupt handling, 153
Inversion, 49
IR_STOPM.SRC, 8
ISTEP, 22

J

Joint coordinate system, 59
Jump instruction, 115

K

Kinematic sequence, 64, 65
Kinematic singularities, 73
KRL assistant, 105
KUKA Robot Language, 105

L

- Limits-TTS, 204
- LIN, 83, 111
- LIN !, 103
- LIN-CIRC approximate positioning, 98
- LIN-LIN approximate positioning, 93
- LIN_REL, 83, 111
- Linear motions, 83
- Linkage editor, 12
- Linking, 12
- Local, 146
- Local data lists, 175
- Logical operations, 48
- Logical operators, 48
- LOOP, 123
- Loops, 118

M

- Machine language, 12
- Main run, 86
- Manual entry, 191
- Masking out bits, 50
- Mechanical zero position, 68
- Merker, 115
- Mirror, 190
- Motion instructions, 59
- Motion parameters, 104
- Motion programming, 59
- Motions with approximate positioning, 89
- MSG_DEMO.SRC, 8
- MSTEP, 22

N

- Names, 29
- NEW_SERV.SRC, 8
- Non-rejecting loop, 121

O

- One-dimensional array, 35
- Operand, 42
- Operator, 42
- Orientation control, 78
- Orientation criterion, 93
- ORing, 49
- Overhead, 74

P

- P00.DAT, 8
- P00.SRC, 8
- Parameter list, 148
- Parameter transfer, 148
- PERCEPT.SRC, 9
- Placeholder, 106
- Point separator, 38
- Point-to-point motions, 66
- POS, 40, 107
- Position specification, 107
- Predefined structures, 40
- Print, 186
- Priority, 51, 156, 165, 169
- Priority of operators, 51
- Program branches, 115
- Program correction, 14
- Program execution control, 115
- Program run modes, 22
- PROKOR, 14
- PSTEP, 22
- PTP, 66, 69, 109
- PTP !, 103
- PTP-CP approximate positioning, 99
- PTP-PTP approximate positioning, 90
- PTP_REL, 69, 109
- PUBLIC, 176
- PULSE, 136
- Pulse outputs, 136

R

- REAL, 33, 34
- Rejecting loop, 120
- Relational operators, 47
- REPEAT, 121
- Replace, 16
- RESUME, 161
- Reverse transformation, 60
- Robot coordinate system, 63

S

- S and T, 73
- Setting bits, 50
- SIGNAL, 128
- Simple data types, 33
- SIN(X), 52
- Sine, Cosine, Tangent, 52

SPS.SUB, 8
SQRT(X), 52
Square root, 52
Start of approximate positioning, 93
Status, 70, 73
STRUC, 38
Structure and creation of programs, 7
Structures, 38
Subprograms, 9, 145
SWITCH, 117
Switch interrupt off, 156
Switch interrupt on, 156
Switching action, 169
Synchronous PTP, 66
System files, 54
System variables, 54

T

TAN(X), 52
TCP, 65
TCP adjust, 196
Teaching points, 103
Three-dimensional array, 37
Timers, 54
Tool Center Point, 66
Tool change, 102
Tool coordinate system, 63
Tool, fixed, 65
Tool-based moving frame, 200
Touch Up, 107
Translations, 62
TRIGGER, 165
Trigger, 165
TTS, 200
Turn, 70, 73
Two-dimensional array, 36

U

UNTIL, 121
USER_GRP.DAT, 9
USER_GRP.SRC, 9
USERSPOT.SRC, 9

V

Value assignment, 29
VAR, 106
Variable + path-related, 80

Variable + space-related, 82
Variable life, 30
Variables and declarations, 29
Variables and names, 29
Velocity, 77
Velocity criterion, 93

W

WAIT, 124
Wait instructions, 124
WEAV_DEF.SRC, 9
WHILE, 120
World coordinate system, 63
Wrist root point, 74